analysis of algorithms

determine how much in the way of resources the algorithm will require.

analysis of algorithms

computing/
cpu time

time complexity

memory/
disk space

space complexity

computing/
cpu time

time complexity

memory/
disk space

space complexity

factors

factors

computer
compiler
algorithm
input

factors

computer
compiler
algorithm
input

estimation of running time

empirical

algorithm analysis

code, then run
along with a
timer.

empirical

```
#iniclude <time.h>
main(){
   clock_t start, end;
   double cpu_time_used;

   start = clock();

   /*Do some stuff here*/

   end = clock();

   cpu_time_used = ((double)(stop-start)) /
   CLOCKS_PER_SEC;
}
```

obtain the expected running time of actual code without actually running it.

algorithm analysis

based on the problem size (input size), what is the time required to solve the problem?

T(n)

a function that maps input size into the time required to solve the problem.

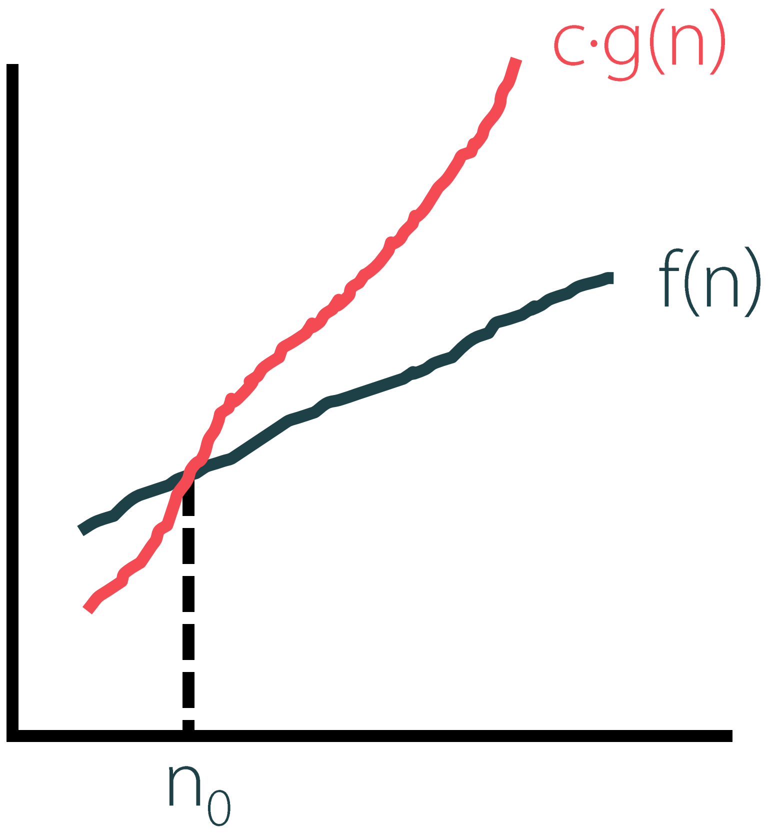
example

$$T(n) \approx O(n^2)$$

example

$T(n) \approx O(n)$

mathematical background



O (Big-Oh)
upper bound

$T(n) = O(f(n))$
if there are constants
c and n_0 such that
$T(n) \le c \cdot f(n)$ when $n \ge n_0$.

O (Big-Oh)
upper bound

Let $T(n) = 1000n$
$f(n) = n^2$

$T(n)$ is larger than $f(n)$ for small values of n, but n^2 grows at a faster rate than $T(n)$.

Therefore, $T(n) \approx O(n^2)$

O (Big-Oh)
upper bound

example

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

$$T(n) = O(\ ?\ )$$

example

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

$$T(n) = O(n^2)$$

$$\text{or } O(n^3)$$

$$\text{or } O(n^4)$$

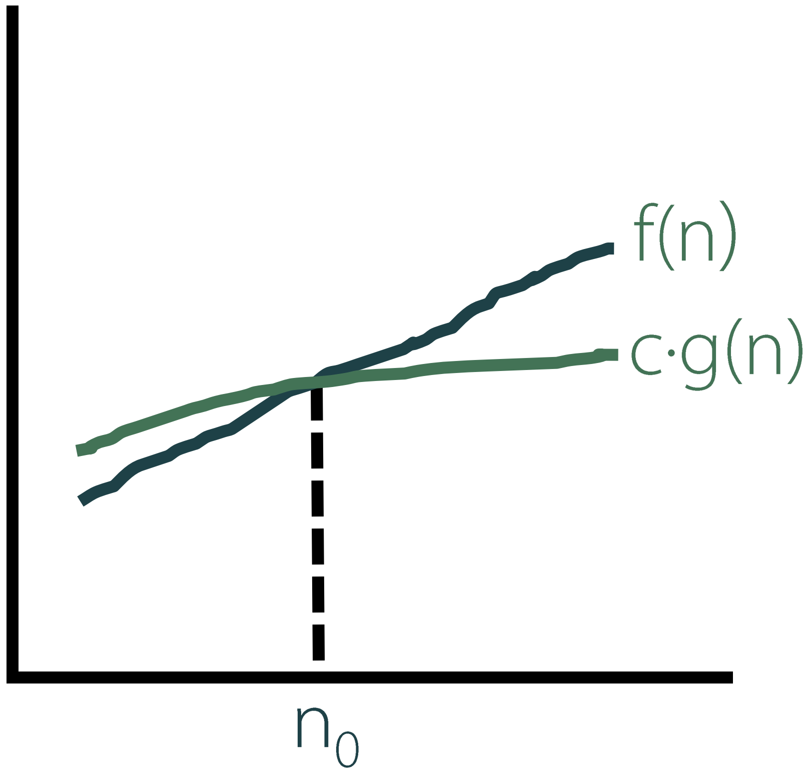
example

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

$$T(n) = O(n^2)$$

$$\text{or } O(n^3)$$

$$\text{or } O(n^4)$$



Ω (Big-Omega)
lower bound

$T(n) = \Omega(g(n))$
if there are constants
c and n_0 such that
$T(n) \geq c \cdot g(n)$ when $n \geq n_0$.

Ω (Big-Omega)
lower bound

θ (Theta)
tight bound

$T(n) = \theta(h(n))$
if and only if
$T(n) = O(h(n))$ and
$T(n) = \Omega(h(n))$.

θ (Theta)
tight bound

lower bound analysis

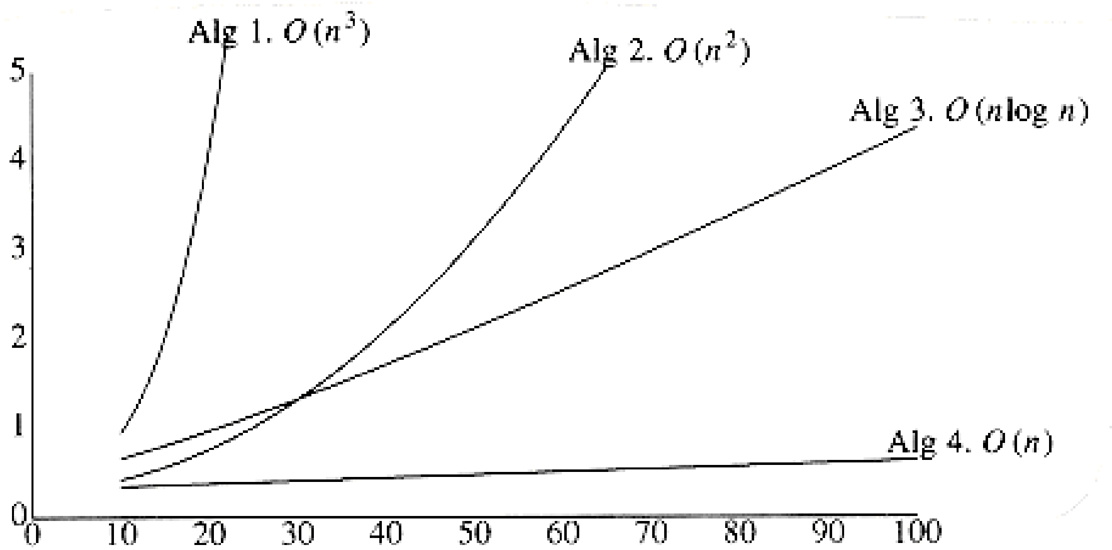
this is used to describe how fast a given problem can be solved.

upper bound analysis

this is used to describe the worst-case performance of an algorithm.

complexity
classes

Big-Oh	description (speed)
$O(1)$	constant
$O(\log n)$	logarithmic
$O(\log^2 n)$	log squared
$O(n)$	linear
$O(n \log n)$	loglinear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$, $k \geq 0$	polynomial
$O(a^n)$, $a > 1$	exponential


Alg 1. O(n³)
Alg 2. O(n²)
Alg 3. O(n log n)
Alg 4. O(n)
5
4
3
2
1
0
0
10
20
30
40
50
60
70
80
90
100

running time calculations

algorithm analysis

calculation of

$$\sum_{i=1}^{n} i^3$$

example

```
int sum(int n){
   int i, sum;
   sum = 0;
   for(i=1;i<=n;i++)
        sum += i*i*i;
   return sum;
}
```

```
int sum(int n){
   int i, sum;
   sum = 0;                  1
   for(i=1;i<=n;i++)         2n+2
        sum += i*i*i;        3n
   return sum;               1    +
}                            ---------
                             5n+4
```

```
int sum(int n){
   int i, sum;
   sum = 0;                  1
   for(i=1;i<=n;i++)         2n+2
        sum += i*i*i;        3n
   return sum;               1    +
}                            -------
                             5n+4
```

$$T(n) = O(n)$$

general rules

at most the running time of the statement inside the loop (including tests) times the number of iteration.

for loops

the running time of the statement multiplied by the product of the sizes of all the for loops.

nested for loops

```
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
     k++;
```

nested
for loops

```
for(i=0;i<n;i++)
   for(j=0;j<n;j++)
        k++;
```

$$T(n) = O(n^2)$$

nested
for loops

the maximum is the one that counts.

consecutive statements

```
for(i=0;i<n;i++)
   a[i] = i+1;

for(i=0;i<n;i++)
   for(j=0;j<n;j++)
      a[i] += a[j];
```

consecutive statements

```
for(i=0;i<n;i++)
   a[i] = i+1;

for(i=0;i<n;i++)
   for(j=0;j<n;j++)
      a[i] += a[j];
```

$T(n) = O(n^2)$

consecutive statements

```
if( cond )
        S1
else
        S2
```

if-else statements

never more than the running time of the test plus the larger of S1 and S2.

if-else statements

examples

```
for(i=0;i<n;i++)
  for(j=0;j<n*n;j++)
    a[i] = i+j;
```

examples

```
for(i=0;i<n;i++)
  for(j=0;j<i;j++)
    s++;
```

examples

```
for(i=0;i<n;i++)
  for(j=0;j<i;j++)
    s++;
```

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

$T(n) = O(n^2)$

examples

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

examples

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

$T(n) = T(n\text{-}1) + 1 = O(n)$

examples

```
for(i=0;i<n;i++)
   if(i%2 == 0)
      for(j=0;j<n;j++)
         s++;
```

examples

```
for(i=0;i<n;i++)
  if(i%2 == 0)
     for(j=0;j<n;j++)
       s++;
```

$$T(n) = (n/2)n = O(n^2)$$

examples

```
for(i=0;i<n*n;i++)
  for(j=0;j<2n;j++)
      s++;
```

examples

```
for(i=0;i<n*n;i++)
   for(j=0;j<2n;j++)
         s++;
```

$$T(n) = n^2 2n = 2n^3 = O(n^3)$$

search algorithms

```
int linearSearch(int a[],int n,int x){

  int i;
  for(i=0;i<n;i++)
    if(a[i] == x) break;
  return (i<n);

}
```

```
a[5] = {2, 7, 8, 2, 3};

linearSearch(a, 5, 2);
```

O(1)

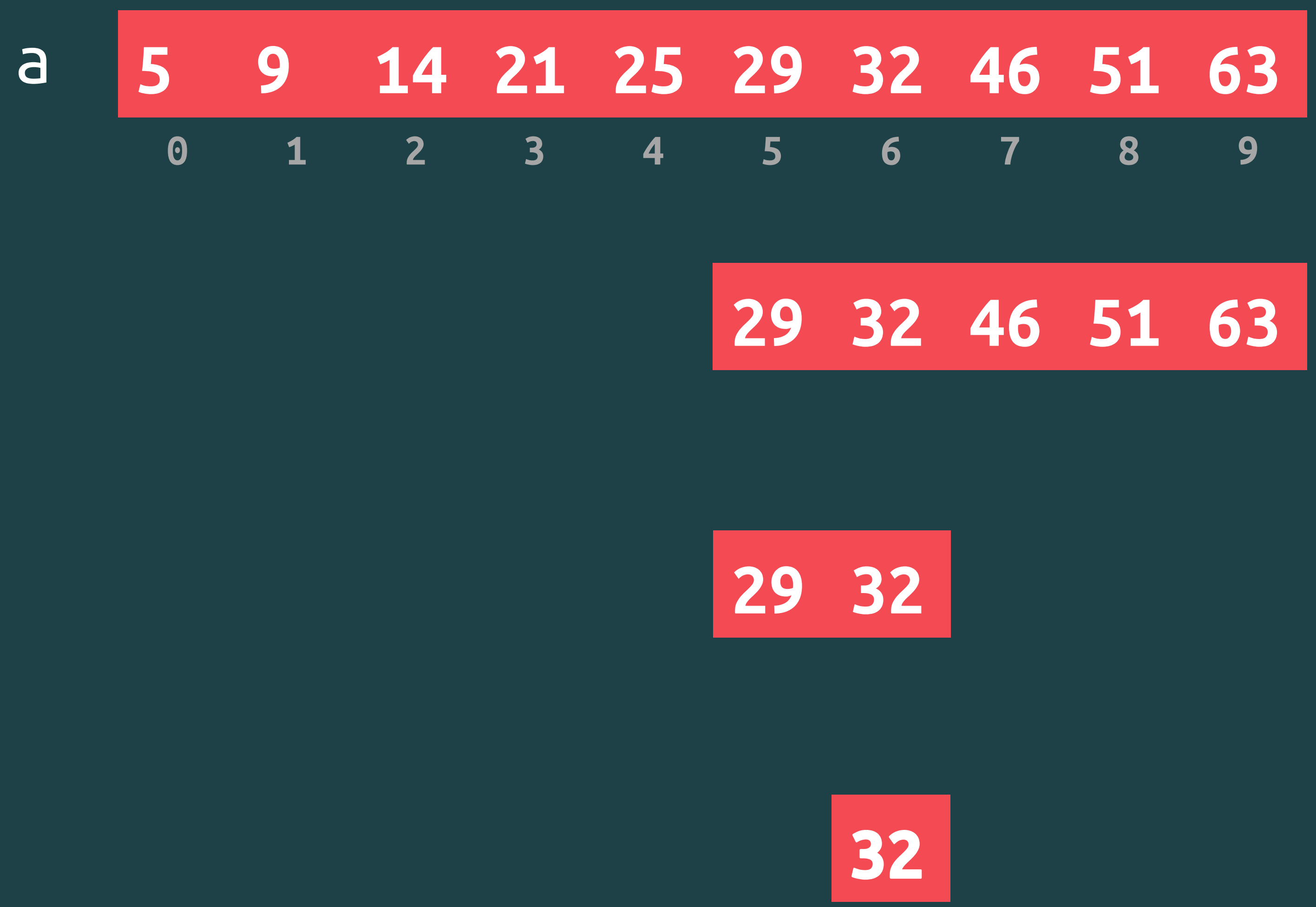
best case complexity

```
a[5] = {2, 7, 8, 2, 3};

linearSearch(a, 5, 3);
```

$O(n)$

worst case complexity

```
int binarySearch(int a[],int n,int x){
  int lower, upper, mid;
  lower = 0;
  upper = n-1;
  while(lower <= upper){
    mid = (lower+upper)/2;
    if(x > a[mid]) lower = mid+1;
    else if(x < a[mid]) upper = mid-1;
    else return 1;
  }
  return (0);
}
```




```
binarySearch(44);
```

An algorithm is O(log n) if it takes constant (O(1)) time to cut the problem size by a fraction (which is usually 1/2).

If constant time is required to merely reduce the problem by a constant amount (such as to make the problem smaller by 1), then the algorithm is $O(n)$.

```
a[5] = {2, 7, 8, 14, 21};

binarySearch(a, 5, 8);
```

O(1)

best case complexity

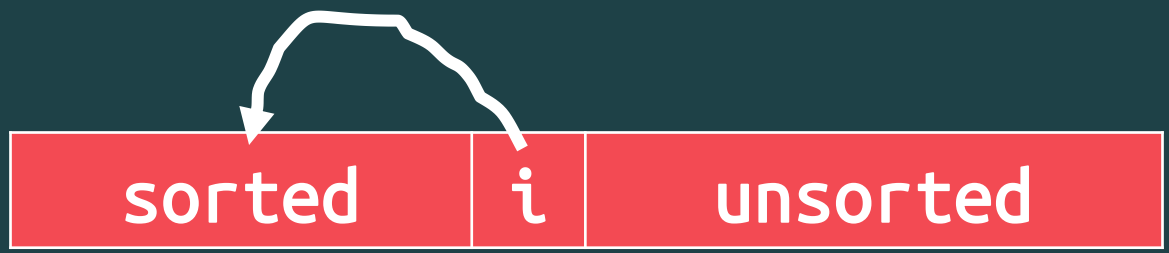
```
a[5] = {2, 7, 8, 14, 21};

binarySearch(a, 5, 5);
```

$O(\log n)$

worst case complexity

sorting
algorithms

```
void insertionSort(int a[],int n){
  int i, j;
  for(i=1;i<n;i++)
    for(j=i;j>0;j++)
      if(a[j]>a[j-1])
        swap(&a[j-1], a[j]);
      else break;
}
```


sorted
i
unsorted

1	5	2	6	4


1 5 2 6 4

1	2	5	6	4

1 2 5 6 4

1 2 4 5 6

The input is inversely sorted.

worst case complexity

a					
	10	9	8	7	6
	0	1	2	3	4

i	# of moves
1	1

a	10	9	8	7	6
	0	1	2	3	4
	9	10	8	7	6

i	# of moves
1	1
2	2

a					
	10	9	8	7	6
	0	1	2	3	4
	9	10	8	7	6
	8	9	10	7	6

i	# of moves
1	1
2	2
3	3

a

0	1	2	3	4
10	9	8	7	6
9	10	8	7	6
8	9	10	7	6
7	8	9	10	6

i	# of moves
1	1
2	2
3	3
4	4

a						i	# of moves
	10	9	8	7	6		
	0	1	2	3	4		
	9	10	8	7	6	1	1
	8	9	10	7	6	2	2
	7	8	9	10	6	3	3
	6	7	8	9	10	4	4 +
						total	10

$$
\begin{aligned}
T(n) &= 1 + 2 + \ldots + n\text{-}1 \\
&= n(n-1)/2 \\
&= (n^2 - n) / 2
\end{aligned}
$$

$O(n^2)$

The input is pre-sorted.

best case complexity

a	6	7	8	9	10	i	# of moves
	0	1	2	3	4		
	6	7	8	9	10	1	1
	6	7	8	9	10	2	1
	6	7	8	9	10	3	1
	6	7	8	9	10	4	$\frac{1 \quad +}{4}$

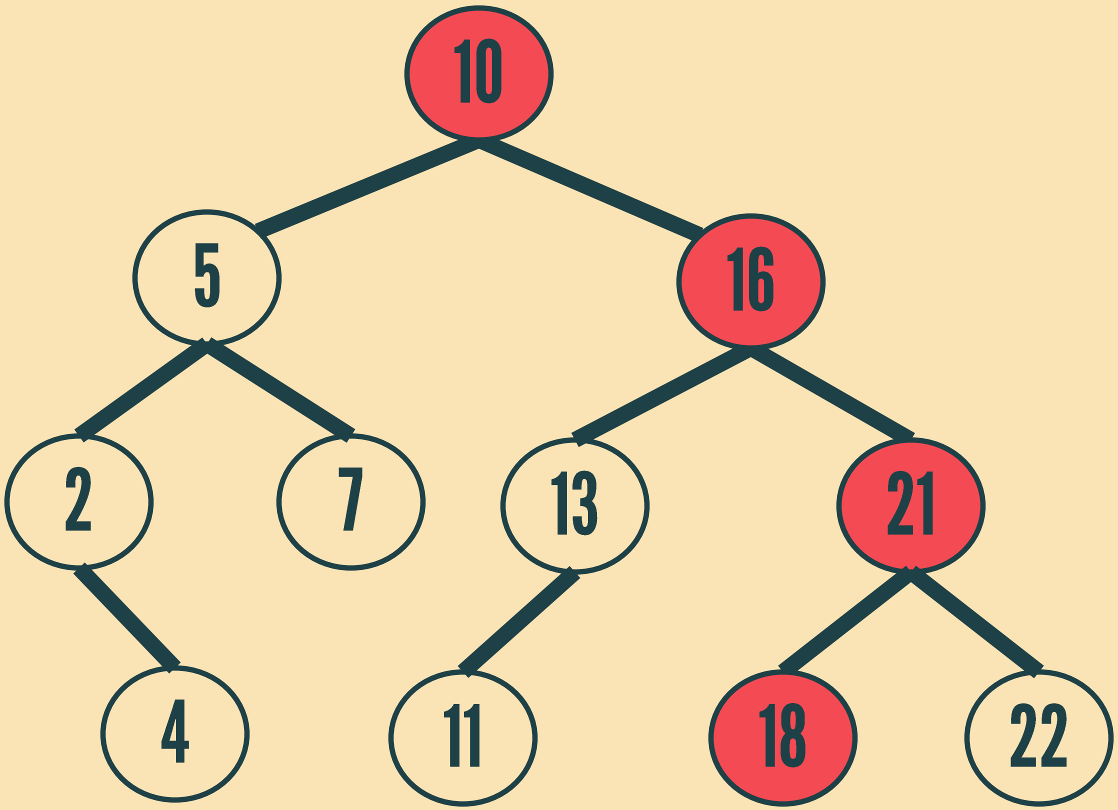
$$T(n) = 1 + 1 + \ldots + 1$$
$$= n-1$$

$$O(n)$$

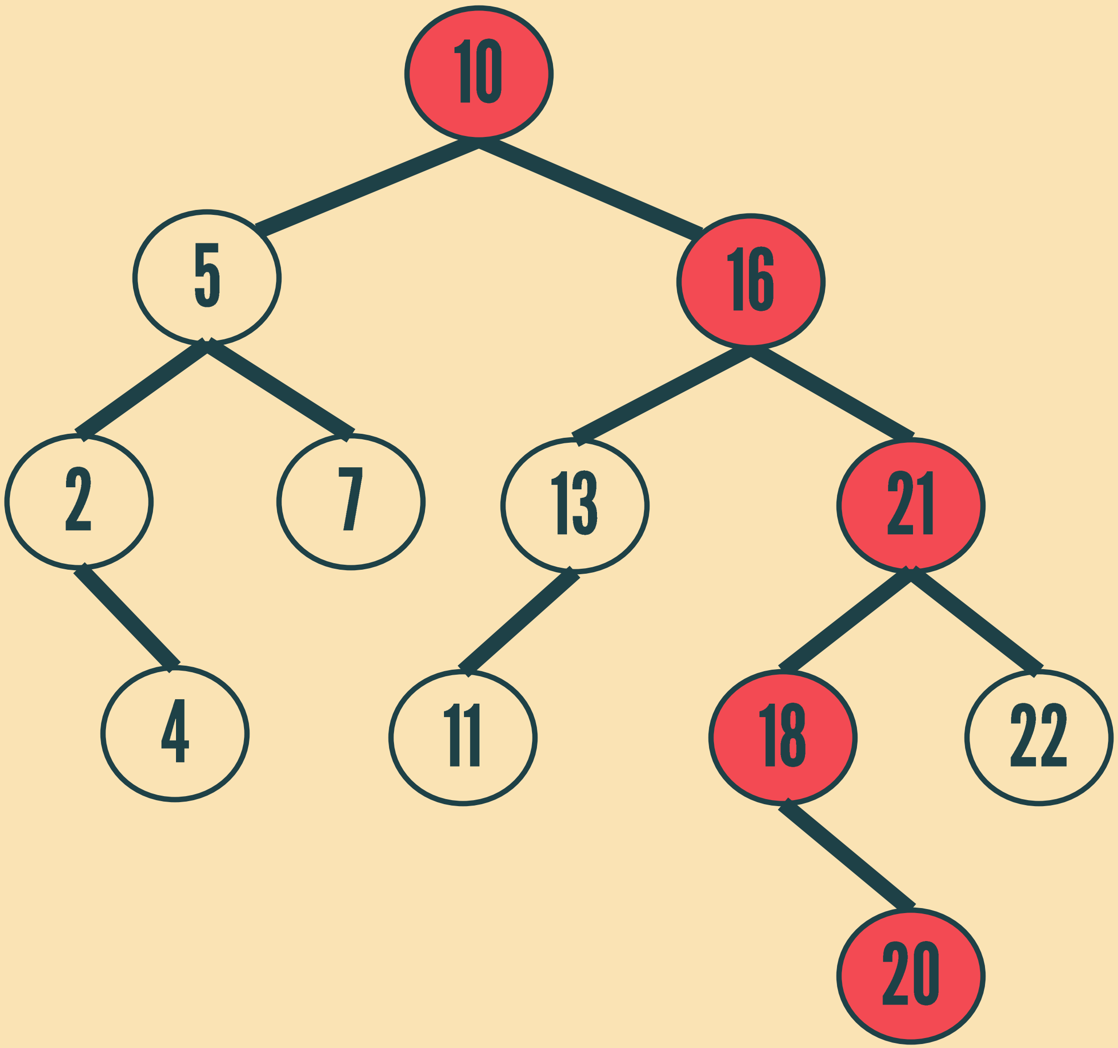
tree-based
algorithms

```
BST* search(BST *root, int x){
  BST *temp = root;
  while((temp!=NULL)&& temp->value!=x){
    if(x < temp->value)
      temp=temp->left;
    else
      temp=temp->right;
  }
  return temp;
}
```

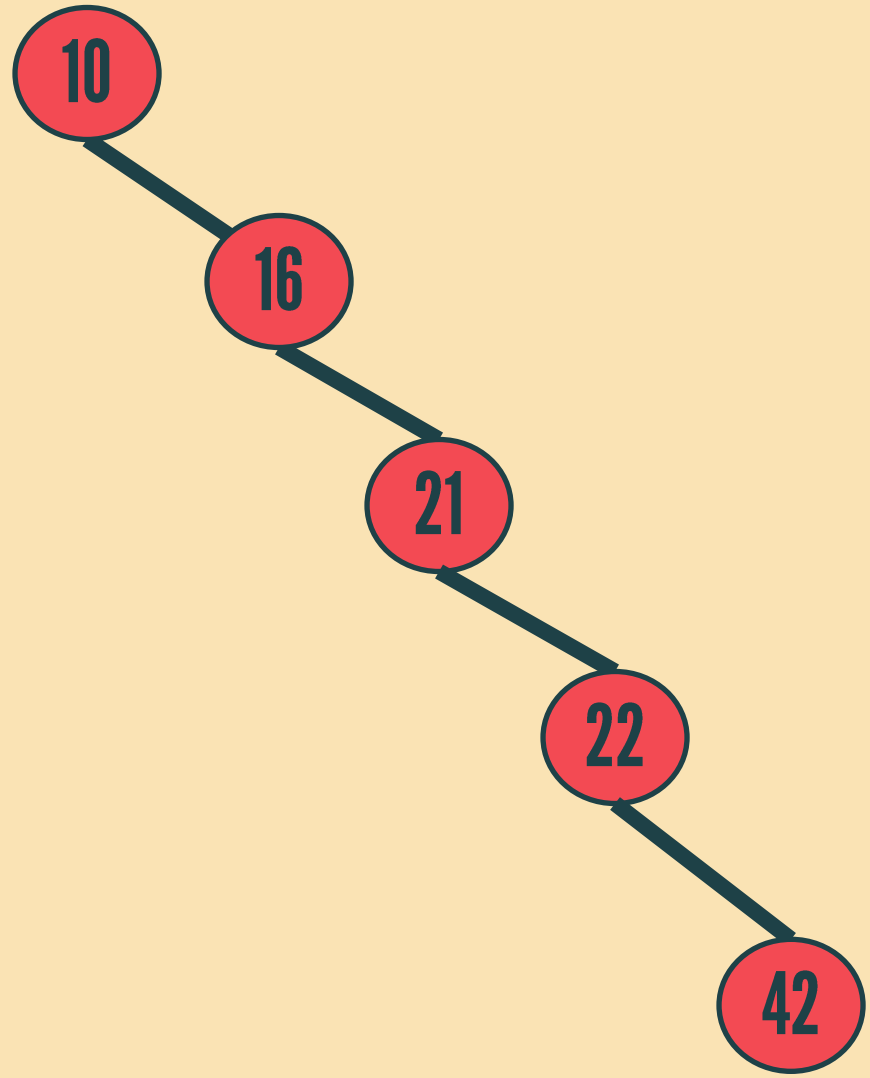
search(18)



```
insert(20)
```

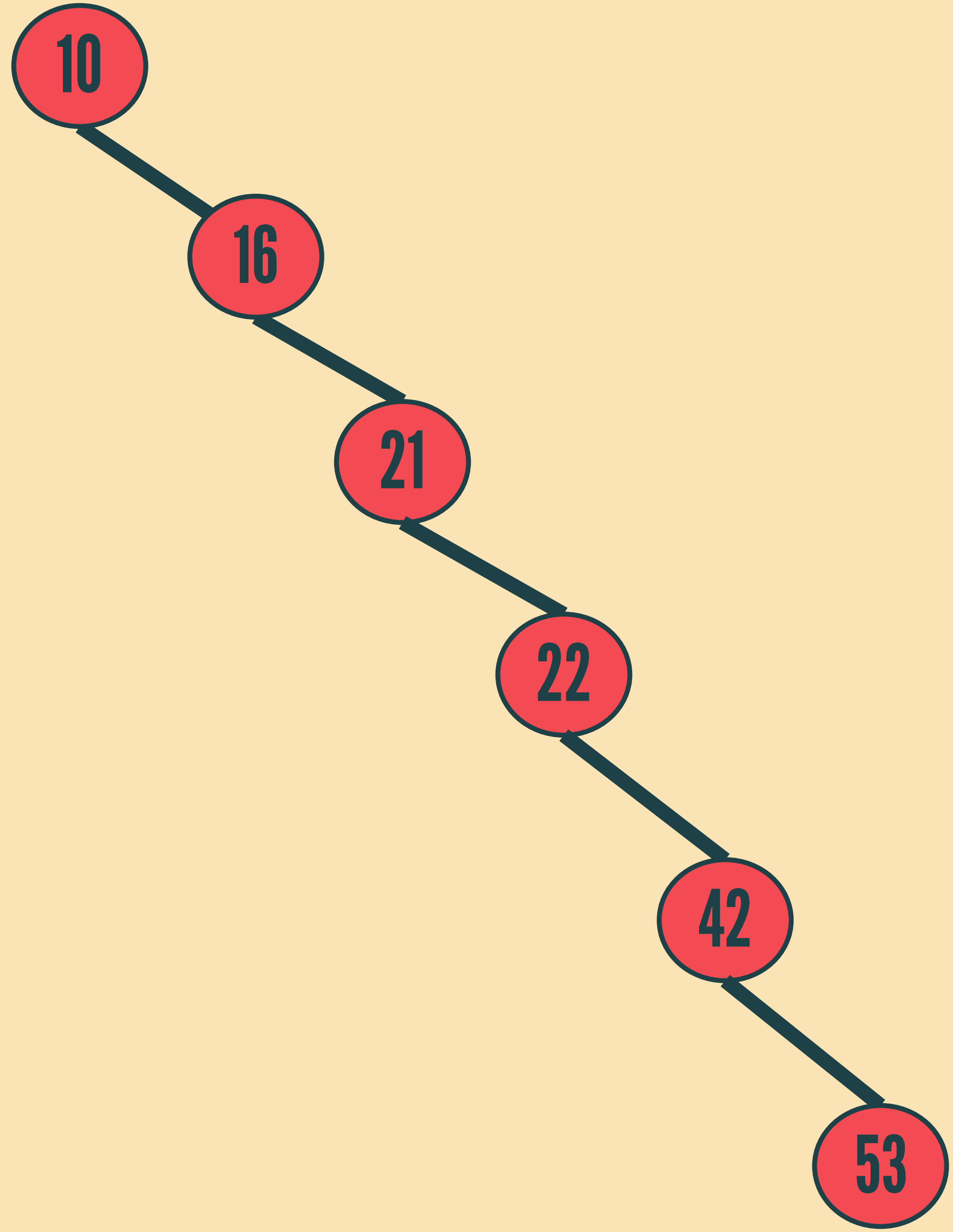


search(42)



worst case complexity

`insert(53)`



worst case complexity

BST
O(n)

AVL
O(log n)