

INTRODUCTION

In this course we study questions such as :

- Given a problem, how do we find an (efficient) algorithm for it ?
- How do we measure the complexity (resource requirements) of an algorithm ?

Problem = set of instances of a question

Solution = set of answers to these instances.

eg. PRIME

is 2 a prime? YES is 3 a prime? YES

is 10^{300} a prime? NO is $2^{2^{2^2}}$ a prime? NO

Slide Lecture 1 -1

- Algorithm: An effective procedure that maps a problem to it's solution.

- effective procedure = "program".

- Problem = question + input parameter(s)

The size of the problem can be characterised by an integer n
E.G.

#inputs (sort)

#digits of input parameter (prime)

Slide Lecture 1 -2

Is there an algorithm for each problem ?

NO

(1) Problem (its question) needs to be effectively specified.

"How many angels can dance on the head of a pin " is not an effectively specified problem.

(2) Even if it is effectively specified, there is not necessarily an algorithm for a problem.

ULAM's problem :

```
given  f(n)= if n=1 then stop else
           if n odd then f(3n+1)
           else f(n/2)
```

Slide Lecture 1 -3

Question: Will $f(n)$ stop ?

$f(1)$ Stop

$f(2) \rightarrow 1$ - stop

$f(3) \rightarrow 10\ 5\ 16\ 8\ 4\ 2\ 1$ stop

$f(4) \rightarrow 2\ 1$ stop

$f(5) \rightarrow$ stop

$f(6) \rightarrow 3 \rightarrow$ stop

$f(7) \rightarrow 22\ 11\ 34\ 17\ 52\ 26\ 13\ 40\ 20\ 10\ 5$ stop

(1) Nobody has ever found an n for which f doesn't stop

(2) Nobody has ever found a proof $:\forall n \in \mathbb{N} : f(n)$ stops

Slide Lecture 1 -4

More general problem : HALTING PROBLEM

Given a program P (written in language L) and an input X will P(x) terminate ?

- We can prove that this problem cannot be solved for very simple languages L.

⇒ ie, HALTING PROBLEM IS UNDECIDABLE

- Given a program P and a specification S does P execute according to S ?

⇒ VERIFICATION PROBLEM IS UNDECIDABLE

Slide Lecture 1 –5

- Given two programs P_1 and P_2 , are they equivalent ?

$$\forall x P_1(x) = P_2(x)$$

⇒ EQUIVALENCE PROBLEM IS UNDECIDABLE.

This does NOT mean that we can't or shouldn't build VERIFICATION PROGRAMS and PROGRAM TRANSFORMERS but it means that those problems in general are unsolvable.

Slide Lecture 1 –6

Assume P decides the halting problem.
Given a program P1 and data D.

$$P(P1, D) = \begin{cases} \rightarrow \text{ YES, halts} \\ \rightarrow \text{ NO, loops} \end{cases}$$

1. Construct Q(P2) that executes by executing

$$Q(P2) = P(P2, P2) = \begin{cases} \rightarrow \text{ YES, halts} \\ \rightarrow \text{ NO, loops} \end{cases}$$

Slide Lecture 1 -7

2. Now construct Q'(P2) which calls Q(P2) = P(P2, P2).

If Q(P2) HALTS then Q'(P2) LOOPS.

If Q(P2) LOOPS then Q'(P2) HALTS.

Slide Lecture 1 -8

3. Now execute $Q'(Q')$

Inside of Q' is embedded $P(Q', Q')$ which determines whether Q' halts or loops when executes on Q' .

But whatever $P(Q', Q')$ says, $Q'(Q')$ at the topmost level is wired to do the opposite!

If $P(Q', Q')$ says $Q'(Q')$ HALTS then $Q'(Q')$ LOOPS

If $P(Q', Q')$ says $Q'(Q')$ LOOPS then $Q'(Q')$ HALTS

A CONTRADICTION.

Slide Lecture 1 –9

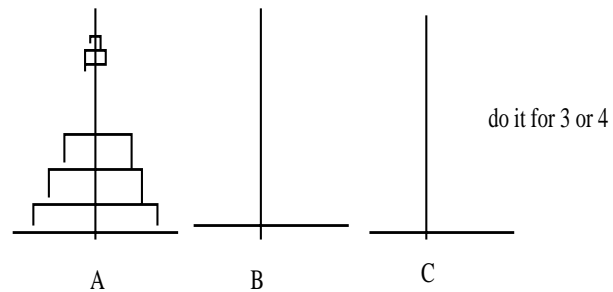
This doesn't mean ALL halting problems are undecidable. But some are. So *in the general case* the halting problem is undecidable: there is no function that *decides* all halting problems.

Slide Lecture 1 –10

INTRACTABILITY

Suppose we have an algorithm for a problem does it terminate in a REASONABLE TIME ?

Example: Towers of HANOI.



Slide Lecture 1 -11

How many *moves** does it take to stack the pieces on pile B, if a bigger piece is never allowed to be pushed on a smaller one ?

MOVE = take top piece off one pile and push on top of another pile; never push a bigger piece on a smaller one.

BUDDIST MONK:

"before it is done the world will have vanished."

Solution: If 1 piece on A

then move it to B.

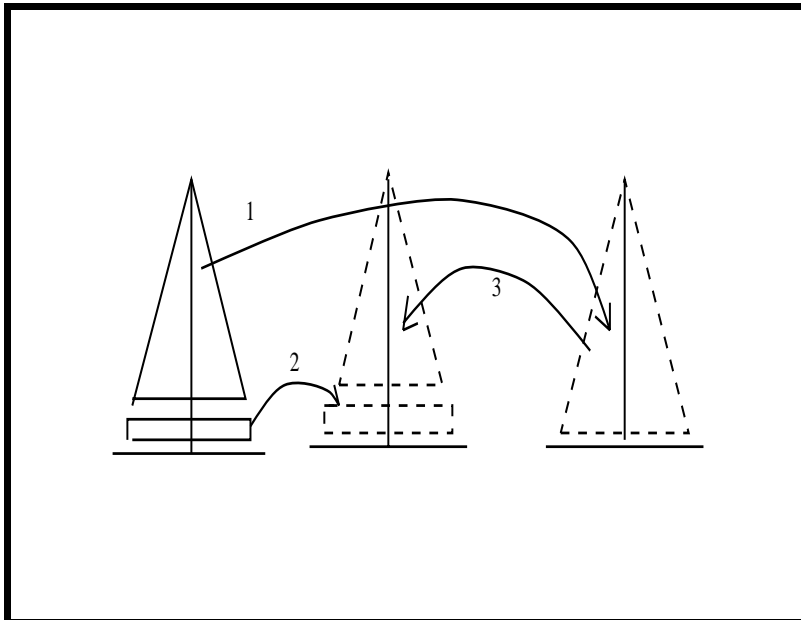
else

(1) move all but the biggest one to C;

(2) move the biggest piece to B;

(3) move the pile from C to B;

Slide Lecture 1 -12



Slide Lecture 1 -13

How many moves ?

$$H(1) = 1$$

$$H(n) = H(n-1) + 1 + H(n-1) = 1 + 2H(n-1)$$

n #moves

1 1

2 3

3 7

4 15

x $2^x - 1$

Was the monk right ? - Allow 1 move per second.

Slide Lecture 1 -14

$$H(100) = 2^{100} - 1 \text{ seconds}$$

$(2^{10} \sim 10^3)$

$$H(100) = 2^{100} - 1 \text{ seconds} \sim 10^{30} \text{ seconds}$$

$$1 \text{ day} \sim 10^5 \text{ seconds} \sim 10^{25} \text{ days} \sim 3 * 10^{22} \text{ years}$$

MORE than the AGE OF THE UNIVERSE !!

- *Could we find a better algorithm ?*

Pile(n-1) MUST be off A and on one stack (say C), before
biggest piece can be moved to its destination.

\implies NO BETTER ALGORITHM

Slide Lecture 1 -15

```
Algorithm HANOI

procedure H(n,r,s)

  if n = 1 then robot (r -> s)
  else { H(n-1, r, 6-r-s);
        robot(r -> s);
        H(n-1, 6-r-s, s);}
  endif
  return
endproc

H(number,Initial-P,Final-P)

end HANOI
```

Slide Lecture 1 -16

THEOREM 3

Tower of Hanoi can always be solved.

Proof:

Let $P(N)$ be the statement that Tower of Hanoi can always be solved for N rings.

$P(1)$. Obvious. Move the ring to the finish pole.

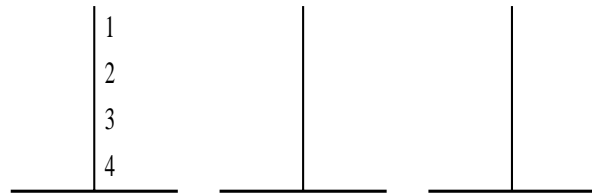
Slide Lecture 1 -17

$P(N) \implies P(N+1)$. If we have $N+1$ rings on a pole, divide this into the problem of moving N rings to the middle pole. Think of the $N+1$ ring as part of the base, since it is larger than any of the previous N rings. Moving the N rings to the middle point is the same as a $P(N)$ Tower of Hanoi problem—which by assumption we can solve. After moving the N rings to the middle pole, move the $N+1$ ring to the finish pole. This is just moving 1 ring and nothing is in the way. Now think of the $N+1$ ring as being the base of the finish pole. It is larger than any of the other N rings and hence does not interfere with moving the other rings. The problem of moving the N rings from the middle point to a position on top of the $N+1$ rings is just another $P(N)$ Tower of Hanoi problem—which by the inductive assumption we can solve. Thus, assuming we can solve the $P(N)$ case, we can also solve the $P(N+1)$ case. QED.

Slide Lecture 1 -18

- Q : Can you find an **ITERATIVE** Algorithm ?

Trace the recursive code.



Slide Lecture 1 -19

```

A[4]B =>
  A[3]C =>
    A[2]B =>
      A-1-C, A-2-B, C-1-B
    A-3-C
    B[2]C =>
      B-1-A, B-2-C, A-1-C
  A-4-B
  C[3]B =>
    C[2]A =>
      C-1-B, C-2-A, B-1-A
    C-3-B
    A[2]B =>
      A-1-C, A-2-B, C-1-B

```

Slide Lecture 1 -20

Pattern :
move smallest ring (1)
move other ring.
repeat Pattern

Now write an iterative HANOI & prove that it works .

- prove will be inductive .

If you study moves of single disks and use this in the algorithm,
check where the whole pile ends up in case of an even number
of rings and an odd number of rings.

Slide Lecture 1 –21



Slide Lecture 1 –22

Algorithm Complexity

is measured in units of time and space.

Example: Search X in telephone book Y.

```
Lin(x,y)
  found:= x == y[1]
  i:=1
  while ((i<n) & (not found)) do
    i:=i+1
    found:= x == y[i]
  end-while
end-lin
```

Slide Lecture 1 -23

Observations

1. n = size of y = Problem size
2. We don't know if X is in Y and, if so, where X is in Y .
Thus, we can only give a WORST CASE or an AVERAGE CASE.
3. We don't know the time for 'atomic actions' (machine dependent). We can only determine the order of magnitude of the complexity.

Slide Lecture 1 -24

In case of Lin

1. In the worst case the while loop-body is executed n times.
2. In the average case the while loop-body is executed $(1/2)n$ times. More Precisely:

$$\sum_{i=1}^n 1/n * i = 1/n * n(n+1)/2 = (n+1)/2$$

Slide Lecture 1 -25

Unit of time

- 1: micro second ? NO, too machine specific.
- 2: machine instruction, Maybe, but still too machine specific.
- 3: pieces(s) of code that take constant time to execute .

Unit of Space :

- 1: bit → very detailed, but sometimes necessary.
- 2: 'location' → place for 1 integer → nicer.
but dangerous !

We can code a whole problem or program in 1 INTEGER
(arbitrary sized), SO we must be careful with space analysis !

Slide Lecture 1 -26

NOTATION : If an algorithm processes inputs of size n in $c(n^2)$ time (where $c = \text{constant}$) we say it is an *order n squared* algorithm notation $\mathcal{O}(n^2)$

$g(n) = \mathcal{O}(f(n))$ if \exists constant c such that $g(n) \leq c(f(n))$
for all but a finite values of n

Worst case complexity :

The **maximal** #steps (#bits or #locations) an algorithm will need for inputs size n .

Slide Lecture 1 -27

Average case complexity :

The **expected** #steps, (#locations) an algorithm will need for inputs size n :

$$\sum_{i \in I_n} P_i * C_i$$

Where P_i is the chance that i occurs and C_i is the complexity given input i . ($i \in I_n \dots$ all inputs of size n)

Slide Lecture 1 -28

Is there a better Algorithm than LINEAR for searching a telephone book?

What does this question mean ?

Better worst case ?

Better avg. case ?

Better \mathcal{O} magnitude ?

(Better constant (multiplicative factor) \leftarrow NO)

Yes. Binary Search.

Because Phone books are sorted.

Slide Lecture 1 -29

The Algorithm

Input: n, [Number of items on the list]
 A(i), [Ordered list of items]
 w [Search word]

Slide Lecture 1 -30

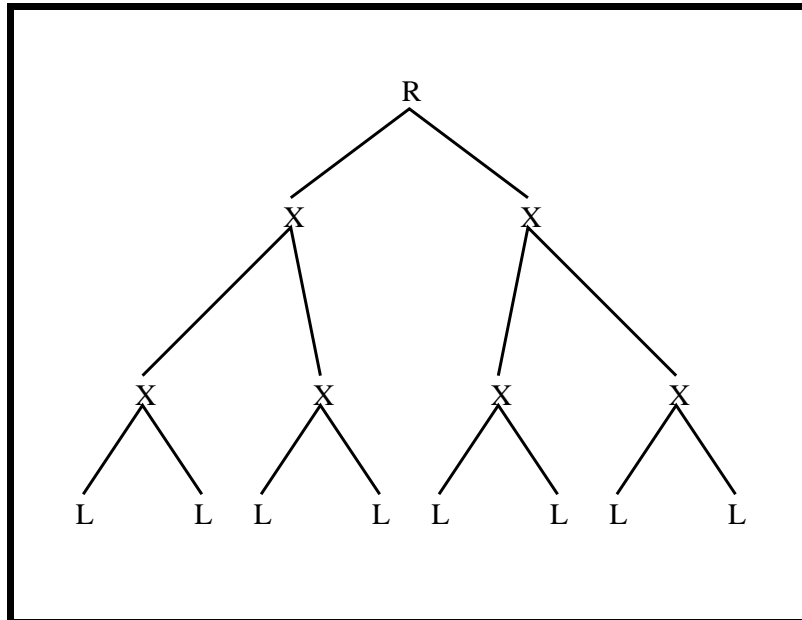
```

ALGORITHM BINSEARCH
  F  $\leftarrow$  1; L  $\leftarrow$  n
  Repeat
    if F > L then output 'failure'; exit
    i  $\leftarrow$  FLOOR((F+L)/2)
    if [Case/Selection structure]
      w = A(i) then output i; exit      [Success]
      w < A(i) then L  $\leftarrow$  i-1
      w > A(i) then F  $\leftarrow$  i+1
    endif
  EndRepeat

Output i                                [Index of word if on list]
      'failure'                          [If word not on list]

```

Slide Lecture 1 -31



Slide Lecture 1 -32

Proof by induction.

Let $P(n)$ be the claim that the worst case complexity for BinSearch is

$$C_n = \lfloor \log_2 n \rfloor + 1$$

Basis:

$$C_1 = \lfloor \log_2 1 \rfloor + 1 = 0 + 1 = 1$$

And a tree/list with 1 element to check takes 1 “compare/test” operation.

Slide Lecture 1 –35

Inductive Step:

When there are N items, the middle one is $\lfloor (N + 1)/2 \rfloor$.

If this is the item we want, we stop. The bound holds.

Otherwise, look at the first or second half.

The length of the first half is

$$\lfloor (N + 1)/2 \rfloor - 1 = \lfloor (N + 1)/2 - 1 \rfloor = \lfloor (N - 1)/2 \rfloor$$

The length of the second half is

$$N - \lfloor (N + 1)/2 \rfloor = N + \lceil -(N + 1)/2 \rceil = \lceil N - (N + 1)/2 \rceil = \lceil (N - 1)/2 \rceil$$

So we have a unsuccessful compare in the middle and then the larger of the two alternatives.

Slide Lecture 1 –36

Thus:

$$C_N = 1 + \max \{ C_{\lfloor (N-1)/2 \rfloor}, C_{\lceil (N-1)/2 \rceil} \}$$

Now, by STRONG INDUCTION, P(k) is true for all $k < N$.

Since $1 + \lfloor \log_2 n \rfloor$ is a monotonically increasing function strong induction implies $C_{\lfloor (N-1)/2 \rfloor} < C_{\lceil (N-1)/2 \rceil}$ which means we can rewrite the bound C_N as follows:

$$C_N = 1 + C_{\lceil (N-1)/2 \rceil} = 1 + \lfloor \log_2 \lceil (N-1)/2 \rceil \rfloor + 1$$

Since $1 = \log_2 2$

$$C_N = 1 + \lfloor \log_2 \lceil (N-1)/2 \rceil \rfloor + \log_2 2$$

$$C_N = 1 + \lfloor \log_2 \lceil (N-1)/2 \rceil + \log_2 2 \rfloor$$

Slide Lecture 1 -37

Since $\log x + \log y = \log xy$

$$C_N = 1 + \lfloor \log_2 2 \lceil \frac{N-1}{2} \rceil \rfloor$$

If N-1 is odd we obtain the desired result:

$$C_N = 1 + \lfloor \log_2 N \rfloor$$

If N-1 is even

$$C_N = 1 + \lfloor \log_2 (N-1) \rfloor$$

but note that when N-1 is even and $N > 1$ that

$$\lfloor \log_2 (N-1) \rfloor = \lfloor \log_2 N \rfloor$$

Slide Lecture 1 -38

because $\lfloor \log_2(N) \rfloor > \lfloor \log_2(N-1) \rfloor$ only when N is an integer power of 2 AND no odd number ($N > 1$) is an integer power of 2.

Thus when $N-1$ is even we also obtain,

$$C_N = 1 + \lfloor \log_2 N \rfloor$$

Slide Lecture 1 –39

CAN WE FIND AN EVEN BETTER ALGORITHM ?

What complexity could that have ?

Ohhh...

Now we have to consider "all possible algorithms"

Slide Lecture 1 –40

NP Completeness

A certain class of problems with Algorithmic Gap

Upperbound: exponential

Lowerbound: polynomial

Trial and Error type Algorithms are the only ones we have found up until now.

Slide Lecture 1 –41

TSP Travelling Salesman Problem

given #cities & distances between them, determine a minimal tour

$$c_1 \rightarrow c_2 \rightarrow c_3 \cdots c_n$$

so that *all* cities occur in tour *only once*.

Algorithm: try all tours & pick the best

Slide Lecture 1 –42

COL Map Coloring Problem

given a Map and a # colors $n: c_1 \cdots c_n$
determine a coloring of countries of the map such that
no adjacent countries have the same color

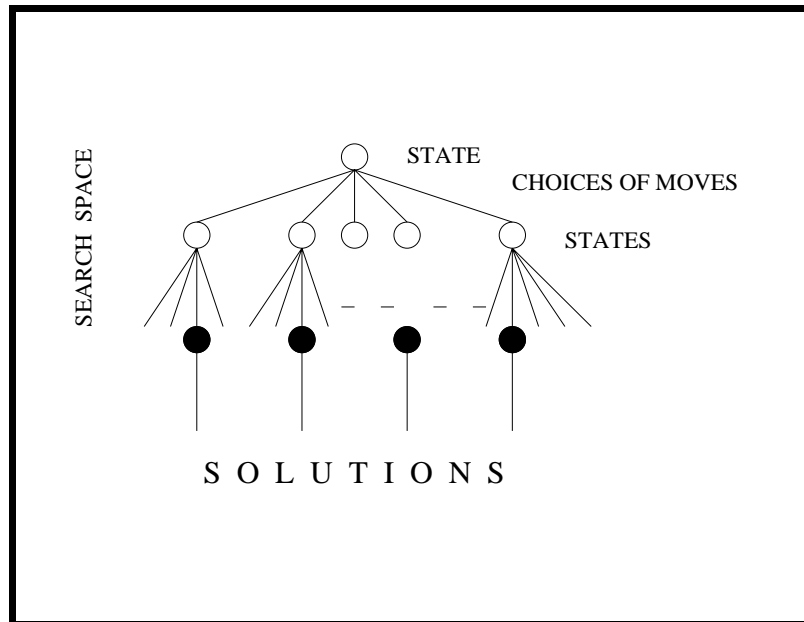
Algorithm: trial and error

SAT

given a boolean expression E with variables $x_1 \cdots x_n$,
determine a truth assignment of values T, F to $x_1 \cdots x_n$
so that E=true

Algorithm: trial and error

Slide Lecture 1 -43



Slide Lecture 1 -44

Backtracking (or trial & error) walks this space in *depth first, left to right fashion*

at choice point: guess & stack the choice point

if solution reached → OK

if clear that no solution is reachable → BACKTRACK:

(POP) go back to last choice point & make a new Guess

Slide Lecture 1 –45

A Non Deterministic Algorithm is one with the ability to select a sequence of right guesses (avoiding the necessity to backtrack)

NP = class of problems solvable in polynomial time by a nondeterministic algorithm

P = class of problems solvable in polynomial time by a deterministic algorithm

All the above mentioned problems (SAT, COL, TSP) are in NP

Q: Would HANOI be in NP?

NO: it really needs to do exponential # moves even if they are all right ones.

Slide Lecture 1 –46

For rather small sizes these problems become INTRACTABLE

$$\text{TSP}(n \text{ cities}) = O(n!)$$

Coping with Intractability

1. **Approximation:** Find a nearly optimal solution for the Euclidean

e.g. TSP we can find a path that is not more than twice the length of the minimal tour in polynomial time

2. **Randomization:** Use probabilistic algorithm (algorithm based on 'coin toss')

Slide Lecture 1 -47

E.g. PRIME

Lin Test (p)

```
for i in 2,3 step 2 to  $\sqrt{p}$ 
  if i divides p then return (NO)
return (YES)
```

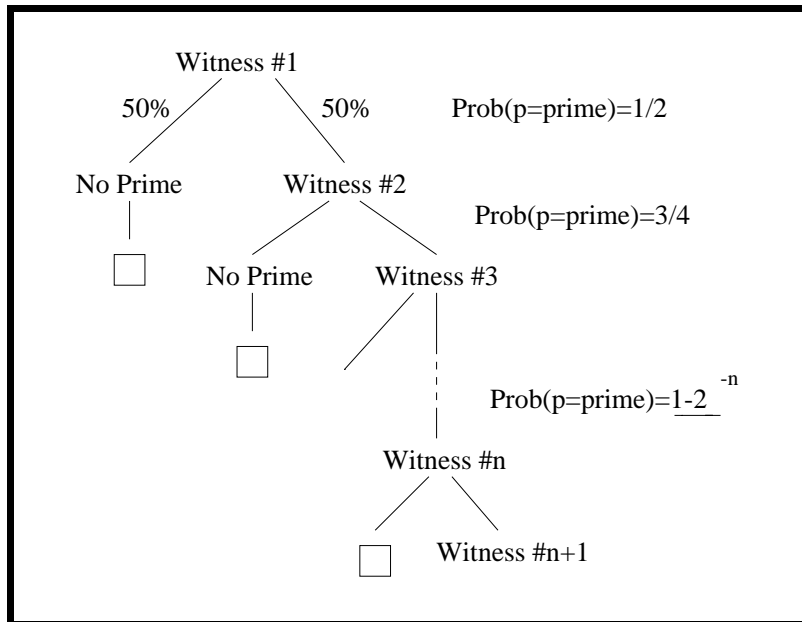
problem each 'chop' only discards one possibility

BETTER SOLUTION: with each *chop* we:

```
either: find p not prime
or: double the probability that p is prime
```

Such a test is called a **witness**

Slide Lecture 1 -48



Slide Lecture 1 -49

Application: *Cryptography*

$$n = p_1 \cdot p_2$$

public key private key

Other applications of randomization

- routing through network
- clash avoidance in ethernet protocols

Slide Lecture 1 -50

Lower bound of a Problem:

Given a set of 'allowed steps' Lower bound = \mathcal{O} (minimal # steps for the worst case of any algorithm solving the problem)

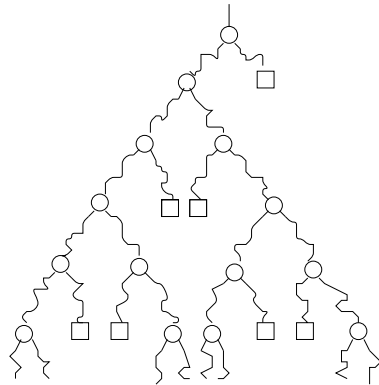
Searching an *ordered array* A for element S

allowed operations (steps):

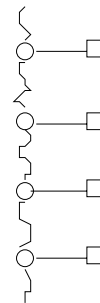
- comparison
- arithmetic on index, other straight line code
- some action on FOUND, NOT FOUND

Slide Lecture 1 -51

Given an array A,
an algorithm is associated with an event tree,
consisting of above steps



e.g. bin



e.g. Lin

Slide Lecture 1 -52

The decision tree will have at least n leaves

Theorem

A binary tree with n leaves has
a height of at least $\log n$

\Rightarrow there is an element that takes at least $\mathcal{O}(\log n)$ steps to be
found in any search algorithm.

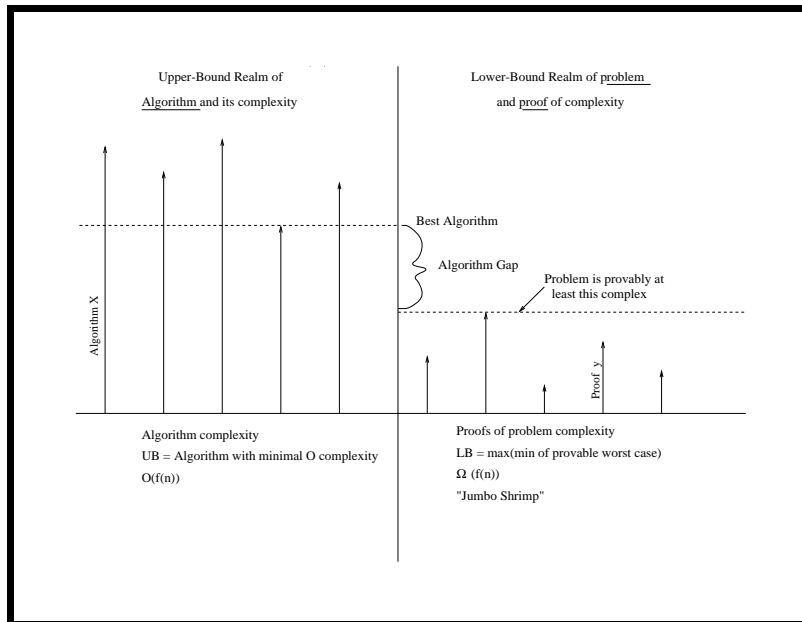
\Rightarrow minimal worst case for search: $\mathcal{O}(\log n)$

\Rightarrow lower bound for search: $\mathcal{O}(\log n)$

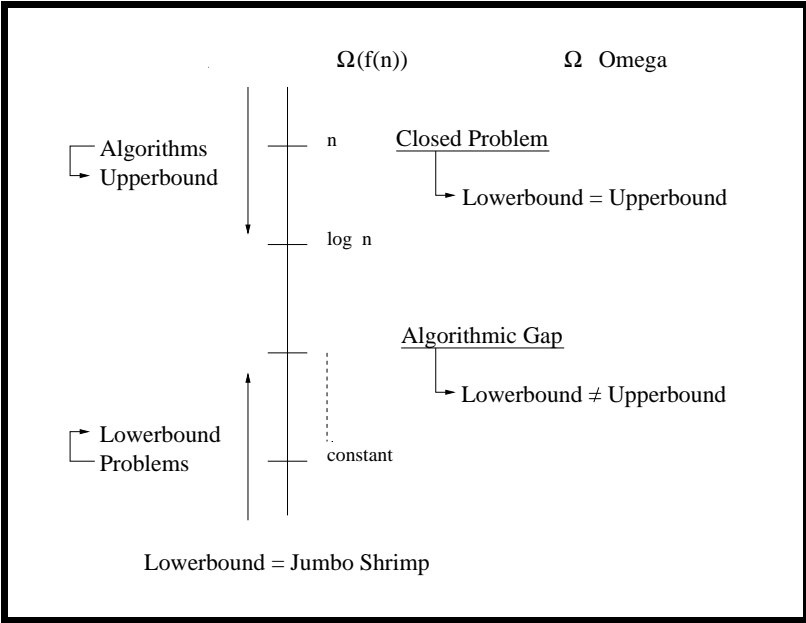
Upperbound = 'best' known algorithm's \mathcal{O} complexity

Lowerbound = max (minimal # steps we can prove the worst
case will take)

Slide Lecture 1 -53



Slide Lecture 1 -54



Slide Lecture 1 -55