Tuesday, March 1, 2022 5:37 AM

Back tracking:

recursion to try "all" possible sol^n (reduce + reuse).

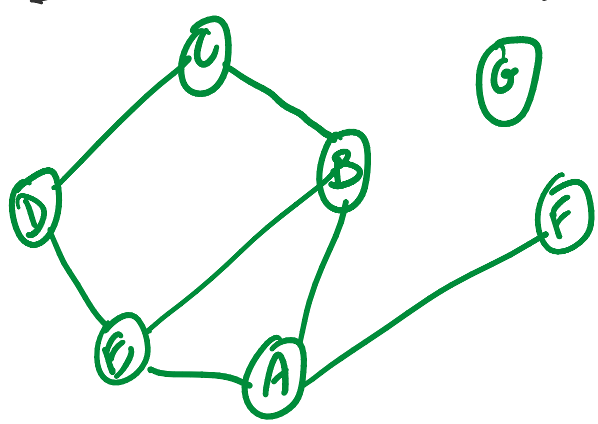
Ex 1: Maximum Independent set.

Given undirected graph $G = (V, E)$ $|V| = n$, $|E| = m$

Find $S \subseteq V$ maximizing $|S|$

s.t. $\forall u, v \in S \Rightarrow uv \notin E$ $\Rightarrow$ S is an independent set

(Optimization Problem)



e.g. $\{A, C\}$ of size 2.

$\{B, D, F\}$ of size 3.

Algo 0: Brute force

Try all subsets. And for each check if indep set

2^n (all subsets), $O(m)$ (check if indep set)

$\Rightarrow O(2^n m)$

$\Rightarrow O(2^n \cdot m)$.

Algo 1: Back tracking

idea: try only "feasible" subsets

Consider a vertex $v \in V$

case I: v is not in opt. sol'n.

remove v & recurse on G - V

case II: v is in the opt. sol'n.

remove v & all of its neighbours AND recurse.

$N(v) = \{u \mid uv \in E\}$

Algo: MIS(G): //return "size" of max. indep. set.

If G is empty return 0.

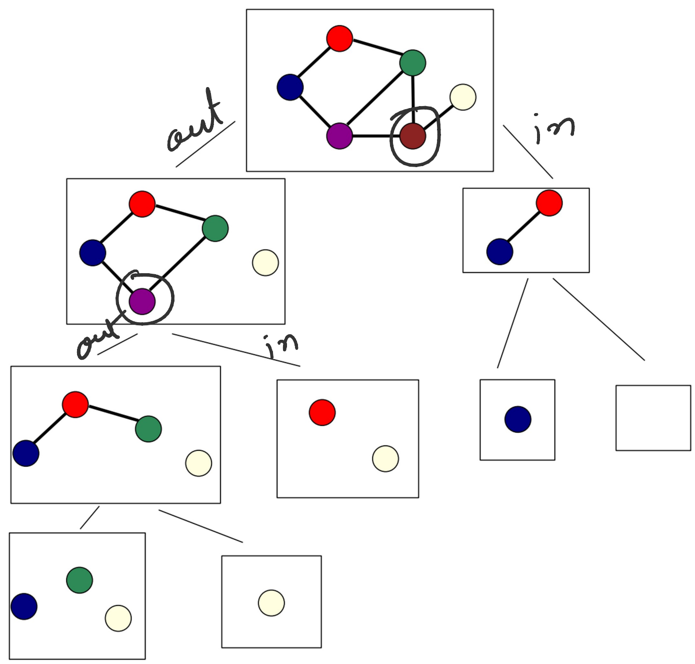
Pick vertex $v \in V$.

If deg(v)=0 then return 1+ MIS(G-V).

return max { MIS(G - V),

1 + MIS(G - V - N(V)) }

$1 + MIS(G - v - N(v)))$



Recursion will automatically backtrack in depth-first search manner.

$\deg(v) = |N(v)|$

Running Time: $T(n) = \underline{T(n-1)} + T(n-1-\deg(v)) + O(m)$

worst-case $\deg(v) \geq 0 \Rightarrow T(n) = 2T(n-1) + O(m)$

$= O(2^n \cdot m)$

Improved Analysis:

$\deg(v) \geq 1 \Rightarrow T(n) = T(n-1) + T(n-2) + O(m)$

Fibonacci #: $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, $F_1 = 1$

suppose $F_n = x^n$

then $x^n = x^{n-1} + x^{n-2}$

$\Rightarrow x^2 - x - 1 = 0$

$\Rightarrow x = \frac{1+\sqrt{5}}{2} \sim 1.618$

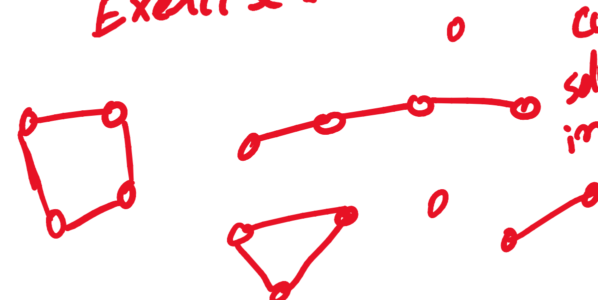
... Ratio

$$\Rightarrow x = \frac{1+\sqrt{5}}{2} \sim 1.618$$
↳ Golden Ratio

$$\Rightarrow T(n) = O(1.618^n \cdot m)$$

Analysis 3: Pick vertex w/ max deg. 4
Stop when deg of every vertex is ≤ 2.

Exercise.



can be solved in $O(n)$

"Greedy"

$$\deg(v) \geq 3 \Rightarrow T(n) = T(n-1) + T(n-4) + O(m)$$

$$x^4 = x^3 + 1$$

$$\Rightarrow O(1.381^n \cdot m)$$

Note: Current Record $O(1.1996^n \cdot m)$
[Xiao & Nagamochi '17]

Ex2: Longest Increasing Subsequence

Given a sequence of numbers

Given a sequence of numbers

$a_1, a_2, \ldots, a_n$

Find sub seq. that maximizes ℓ.

$i_1 \leq i_2 \leq \ldots \leq i_\ell$ s.t. $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_\ell}$

eg. 4, 2, 3, 13, 1, 10, 5, 4, 9, 7, 12

Algo 0 : Brute force

Try all possible subseq. & check

subseq. ← 2^n ; check ← $O(n)$

$\Rightarrow O(2^n \cdot n)$

Algo I : Back tracking

Consider a_n

Case I: a_n is not in the opt. sol^n.

recurse on $a_1, \ldots, a_{n-1}$

Case II: a_n is in the opt. sol^n.

recurse on $a_1, \ldots, a_{n-1}$ &

make sure to pick ele. $\leq a_n$

largest possible number in sol^n = Extra bit of information to be passed to the function.

X) // returns length of longest

number in set → passed ...

LIS($\langle a_1, \ldots, a_n \rangle$, X) // returns length of longest inc. subseq. of ele $\le X$.

If $n=0$ return 0

If $a_n \le X$ then

return max { LIS($\langle a_1, \ldots, a_{n-1} \rangle$, X),

1 + LIS($\langle a_1, \ldots, a_{n-1} \rangle$, a_n) }

else return LIS($\langle a_1, \ldots, a_{n-1} \rangle$, X)

call: LIS($\langle a_1, \ldots a_n \rangle$, ∞)

Naive Analysis: $T(n) = 2T(n-1) + O(1)$

$\Rightarrow O(2^n)$!

"Key Observation": How many "distinct" sub problems?

LIS(para 1, para 2)

para 1 ↓ list of ele.; para 2 ↑ upperbound on picked ele.

distinct para 1 = # prefixes = n.

distinct para 2 = # element in seq. + 1 = $n+1$.

Total = $n \cdot (n+1)$ subproblems.

||.

Total = ...

⇓

Solving same subproblems over & over again!

Avoid this by remembering answers.

↳ **Memoization**

⇒ dynamic programming.

Memoization ver 1: (recursive).

$a_{n+1} = \infty$

LIS(i, j) : // input $\langle a_1, \ldots a_i \rangle$, $x = a_n$.

$L[i,j] = \text{undef}$ $\forall i,j \leq n+1$

If $i=0$ return 0. ←

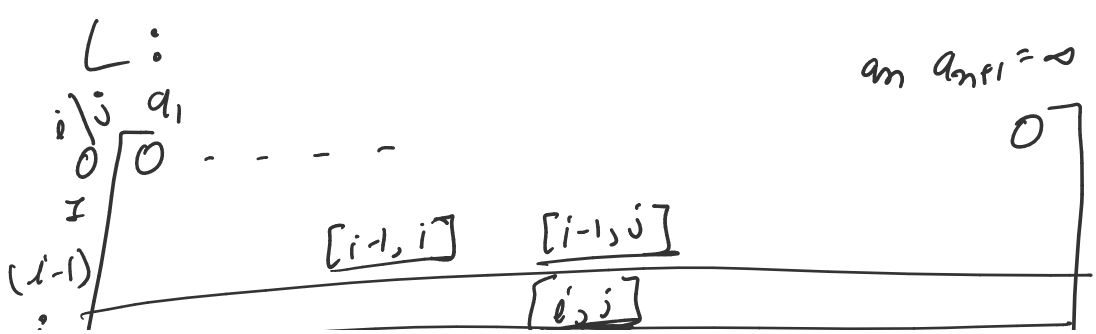
If $L[i,j] \neq \text{undef}$ return $L[i,j]$ ←

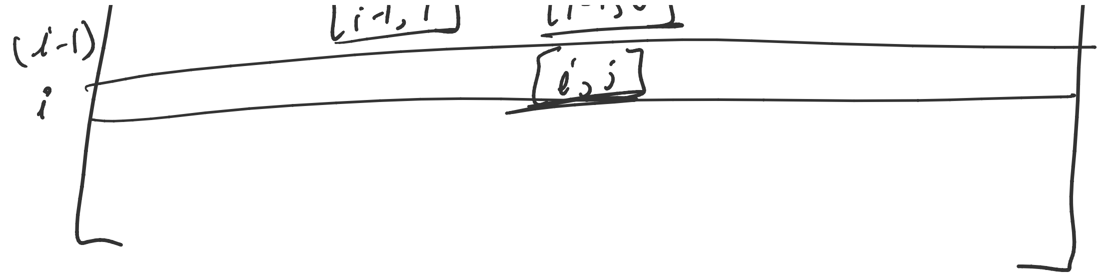
If $a_i \leq a_j$

return $L[i,j] = \max\{ LIS(i-1, j),\ 1 + LIS(i-1, i) \}$

Else return $L[i,j] = LIS(i-1, j)$

⇒ Running time: $O(n^2)$





Memoization Ver 2: (iterative)

idea: start $i=0$ & complete the table row-by-row.

$O(n) \rightarrow$ For $j=1$ to n do $L[0,j] = 0$.

$n \times O(n) \rightarrow$ For $i=1$ to n do

    For $j=1$ to n do

        if $a_i \leq a_j$ $\quad L[i,j] = \max\{L[i-1,j],\ 1+L[i-1,i]\}$

        Else $\quad L[i,j] = L[i-1,j]$

Running time: $O(n) + O(n^2) = O(n^2)$.

Dynamic Programming