Lecture 2020-03-11

Thursday, 11 March, 2021 10:53

Divide-and-Conquer $T(n) \le \text{poly}(n) + aT(\frac{n}{b}) + cT(\frac{n}{d}) + \dots$

$= \text{poly}(n)$

Recursive Backtracking $T(n) \le \text{poly}(n) + aT(n-b) + cT(n-d)$

typically exponential

Today: Dynamic Programming

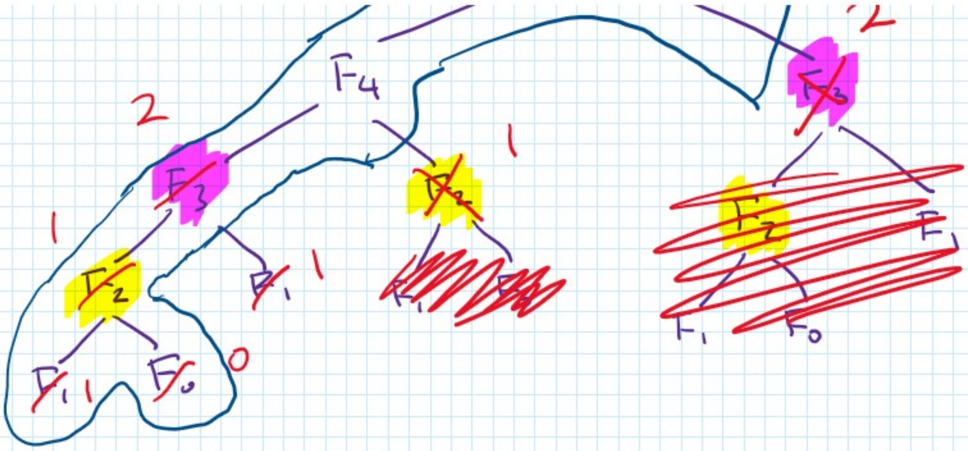
↳ often (but not always) converts special types of recursive backtracking into polynomial time iterative algorithm

Fibonacci numbers

$$F_n = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ F_{n-1} + F_{n-2} & n>1 \end{cases}$$

```
RecFibo(n):
    if n=0:
        return 0
    else if n=1:
        return 1
    else
        return RecFibo(n-1) + RecFibo(n-2)
```

→ RecFibo(n) actually only



actually only $\underline{n}$ recursive subproblems

many of which are computed (recursively) multiple times.

<u>Memoization</u> <u>remember</u> past work.

Q: what data structure? ~~hash map~~ $\boxed{\text{array}}$

```
MemoizedFibo(n):
    if n=0:
        return 0
    else if n=1:
        return 1
    else
        if F[n] is defined
            return F[n]
        else
            F[n] ← MemoizedFibo(n-1) + MemoizedFibo(n-2)
            return F[n]
```

cache miss!

running time?
when is F[n] defined?
↳ how many recursive subproblems
× amount of time to evaluate recursive subproblem.

n subproblems
$\times\ \Theta(n)$ per problem

$\Theta(n^2)$ running time

$F_n = \Theta(\underline{\phi}^n)$

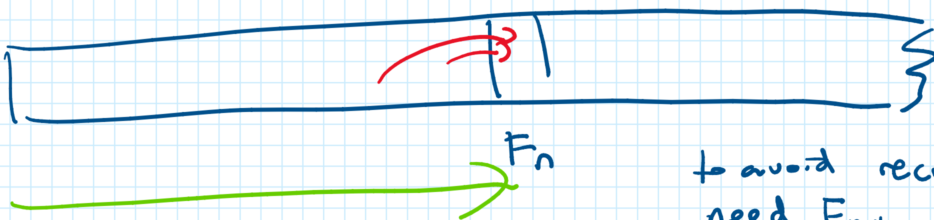
≈ 1.6something

to write down F_n takes $\Theta(n)$ bits / words

$\Theta(n)$ time to write

Dynamic Programming: remove all recursive calls by evaluating & memoizing in a "good" order

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$



F_n

precompute from left to right

to avoid recursive calls, need F_{n-1} & F_{n-2} to already be computed so can do lookups when computing F_n.

DPFibo(n):

```
    F[0] ← 0
    F[1] ← 1
    for i from 2 to n:
        F[i] ← F[i-1] + F(i-2)
    return F(n)
```

$\underbrace{F[i-1] + F(i-2)}_{\Theta(i)}$ ← subtle point

(space efficient: just remember two variables)

Text Segmentation

$\text{IsStrInL}^*(i) = \text{TRUE}$ iff $w[i..n]$ is splittable ($\in L^*$)

$$= \begin{cases} \text{TRUE} & i > n \\ \bigvee_{j=i}^{n} \left(\text{IsStr}(w[i..j]) \wedge \text{IsStrInL}^*(j+1) \right) & \text{o.w.} \end{cases}$$

```
IsStrInL*(i):
  if i > n:
     return TRUE
  else
    for [j from i to n]
      if IsStr(w(i..j))
        if [IsStrInL*(j+1)]
          return TRUE
  return FALSE
```

BOT|HEARTH|AND|SATURNSPIN

FALSE

HEARTH|AND|SATURNSPIN

Same subproblem as before!

memoizing:
what data structure?
array:
SplitTable[i]
= TRUE iff $w[i..n] \in L^*$

evaluation order?
SplitTable[i] depends on SplitTable[j] for all $j > i$.
from n down to i.

Q: what are / how many recursive subproblems?

$O(n)$ $\quad \text{IsStrInL}^*(j)$

$1 \le j \le n+1$

Q: how long to evaluate each one

$O(n)$

by memoizing: $O(n^2)$ time.

```
DP IsStrInL*(~~i~~ w[1..n])
  ~~if i > n~~
    ~~return TRUE~~
  for j from i to n
    SplitTable[i] ← FALSE
```

note: no base case

~~return TRUE~~

for j from i to n
  SplitTable[j] ← FALSE
SplitTable[n+1] ← TRUE ← sets up base case
for j from n down to ~~x~~: 1
  // evaluate SplitTable[j]
  for k from j to n:
    if isStr(w[j..k])
      if SplitTable[k+1]
        SplitTable[j] ← TRUE
return SplitTable(~~x~~ 1)

for loop on n numbers

for loop const

$O(n^2)$

Dynamic Programming

1. Recursive Backtracking
2. What are the recursive subproblems?
3. memoization data structure (often an array)
4. evaluation order
5. which entry to return?

x

Longest Increasing Subsequence

LISGreater(i, k) = length of LIS of $A[i..n]$ consisting of #s $> k$

LISGreater(i, k) = length of LIS of $A[i..n]$ consisting of #s $> k$.

(return LISGreater$(1, -\infty)$)

$$= \begin{cases} 0 & i > n \\ \text{LISGreater}(i+1, k) & A[i] \le k \\ \max \begin{cases} 1 + \text{LISGreater}(i+1, A[i]) \\ \text{LISGreater}(i+1, k) \end{cases} & A[i] > k. \end{cases}$$

Q: parameters of memoization ds?

1d array? no, two parameters.

which are i & k?

Bounds: $1 \le i \le n$. on k? unknown!

equivalent recursive backtracking

LISGreater2(i, j) = length of LIS of $A[i..n]$ consisting of #s $> A[j]$

recursive subproblems: $1 \le i, j \le n+1$

bounded params

→ memoize into 2d array indexed $(O(n) \times O(n))$ by i & j.

$$= \begin{cases} 0 & i > n \\ \text{LISGreater2}(i+1, j) & A[i] \le A[j] \\ \max \begin{cases} 1 + \text{LISGreater2}(i+1, i) \\ \text{LISGreater2}(i+1, j) \end{cases} & A[i] > A[j] \end{cases}$$

eval order?

LISGreater2$[i][j]$ depends on ...

LISGreater2$[i+1][\ldots]$

some j where $j < i$?

DPLIS$(A[1..n])$

DPLIS(A[1..n])

```
A[0] ← -∞
for j from 0 to n
    LISGreater2[n+1][j] ← 0
for i from n down to 1:
    for j from 0 to i:
        // fill in LISGreater2[i][j]
return LISGreater2[1][0]
```

eval order.