Divide-and-Conquer: \[ T(n) \leq \text{poly}(n) + aT\left(\frac{n}{b}\right) + cT\left(\frac{n}{d}\right) + \ldots \]
\[ = \text{poly}(n) \]

Recursive Backtracking: \[ T(n) \leq \text{poly}(n) + aT\left(n-b\right) + cT\left(n-d\right) \]

Today: Dynamic Programming

Often (but not always) converts special type of recursive backtracking into polynomial time iterative algorithm.

Fibonacci numbers:

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

RecFibo(n):

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

\[ \Rightarrow \text{RecFibo}(n) \text{ actually only} \]
Recursively subproblems actually only many of which are computed (recursively) multiple times.

**Memoization**

Q. what data structure? hash map array

Memoized Fibo(n):

1. if n = 0:
   return 0
2. else if n = 1:
   return 1
3. else if F[n] is defined
   return F[n]
4. else
   F[n] ← Memoized Fibo(n-1) + Memoized Fibo(n-2)
   return F[n]

Running time?

when is F[n] defined?

> how many recursive subproblems

× amount of time to evaluate recursive subproblem.

O(n^2) running time

\[ F_n = \Theta(\phi^n) \]

≈ 1.6 something

to write down

F_n takes Θ(n) bits/words

Θ(n) time to write

The cache miss!
Dynamic Programming: remove all recursive calls by evaluating & memoizing in a "good" order.

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

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

DP Fib(n):

- \( F[0] = 0 \)
- \( F[1] = 1 \)

for \( i \) from 2 to \( n \):

\( F[i] = F[i-1] + F[i-2] \)

return \( F(n) \)

(\( \Theta(i) \) a subtle point)

\[ \times \]

Text Segmentation
Is.Str.InL^x(i) = True if \( w[i \ldots n] \) is splittable \((\in L^x) \)

\[
= \begin{cases} 
  \text{True} & \text{if } i > n \text{ return True} \\
  \lor_{j=i}^{n} (\text{Is.Str.In}(w[i \ldots j]) \land \text{Is.Str.In}(j\ldots n)) & \text{else} \\
  \text{If Is.Str.In}(w[i \ldots j]) \land \text{Is.Str.In}(j\ldots n) & \text{return False} \\
\end{cases}
\]

Q: What are/how many recursive subproblems?
\( O(n) \) \( \text{Is.Str.In}^x(j) \)

1 \( \leq j \leq n+1 \)

Q: How long to evaluate each one?
\( O(n) \)

by memory: \( O(n^2) \) time.

\[
\text{DP Is.Str.InL^x(i) : w[i \ldots n]} \
\text{if } i > n \text{ return True} \\
\text{for } j \text{ from } i \text{ to } n \text{ check } \text{Is.Str.In}(w[i \ldots j]) \land \text{Is.Str.In}(j\ldots n) \\
\text{ \( n \cdot n \) base case}
\]
for i from 0 to n
  SplitTable[i] = TRUE
for j from n down to 1
  if isStr(wo[i...j])
    if SplitTable[j+1]
      SplitTable[j] = TRUE
  return SplitTable[1]

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?

Longest Increasing Subsequence

LISgrade(i, k) = length of LIS of A[i...n] consisting of #s \geq k
LISgreater \( (i, k) \) = \text{length of LIS of } A[1..n] \text{ consisting of } #s \geq k.

\[
\begin{cases}
0 & i \geq n \\
\text{LISgreater}(i+1, k) & A[i] \leq k \\
\max \{1 + \text{LISgreater}(i+1, A[i]), \text{LISgreater}(i+1, k)\} & A[i] > k.
\end{cases}
\]

Q: Parameters & memoization do's?

1d array? no, two parameters.

which are i & k?

Bounds? \(1 \leq i \leq n\), on k? \(\text{unknown}\).

equivalent recursive backtracking

\[
\text{LISgreater2}(i, j) = \text{length of LIS of } A[1..n] \text{ consisting of } #s > A[j].
\]

# recursive subproblems:

\(1 \leq i, j \leq n+1\)

bounded params

⇒ memory into

2d array indexed \((O(n) \times O(n)) \text{ by } 1 \& j\).

eval order?

\[
\text{LISgreater2}[i][j] \text{ depends on...} \\
\text{LISgreater2}[i+1][j] \text{ [...] some } j \text{ where } j < i?
\]

DPLIS \((A[1..n])\)
DPLIS (A[1..n])
A(0) \leftarrow -\infty
for i from 0 to n:
    LIS[2][i+1][i] \leftarrow 0
for i from n down to 1:
    for j from 0 to i:
        \text{eval order.}
    // fill in LIS[2][i][j]
return LIS[2][1][0]