Algorithms

Backtracking (really this time...)

Recap

- HW1 due
- HW2: posted
  - oral grading Thursday & Friday morning
  - Sign-ups - Monday

Last time

Finished our intro to recursion

Note that this is not the only technique you'll see!

Just an important one.

We saw several simpler ones:

- sorting
- Hanoi

& more complex:

- multiplication
- linear selection

Another recursive strategy: Backtracking (Chapter 2)

Idea: Build up a solution iteratively.

Setting: an algorithm needs to try multiple options.

Strategy: Make a recursive call for each possibility.

Downside: SLOW

But: - sometimes needed

- often we can speed things up

First two in book:

N queens

```
PLACEQUEENS(Q[1..n], r):
  if r = n + 1
      print Q[1..n]
  else
      for j ← 1 to n
          legal ← TRUE
          for i ← 1 to r − 1
              if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)
                  legal ← FALSE
          if legal
              Q[r] ← j
              PLACEQUEENS(Q[1..n], r + 1)        ⟨⟨Recursion!⟩⟩
```

Figure 2.2. Gauss and Laquière's backtracking algorithm for the n queens problem.

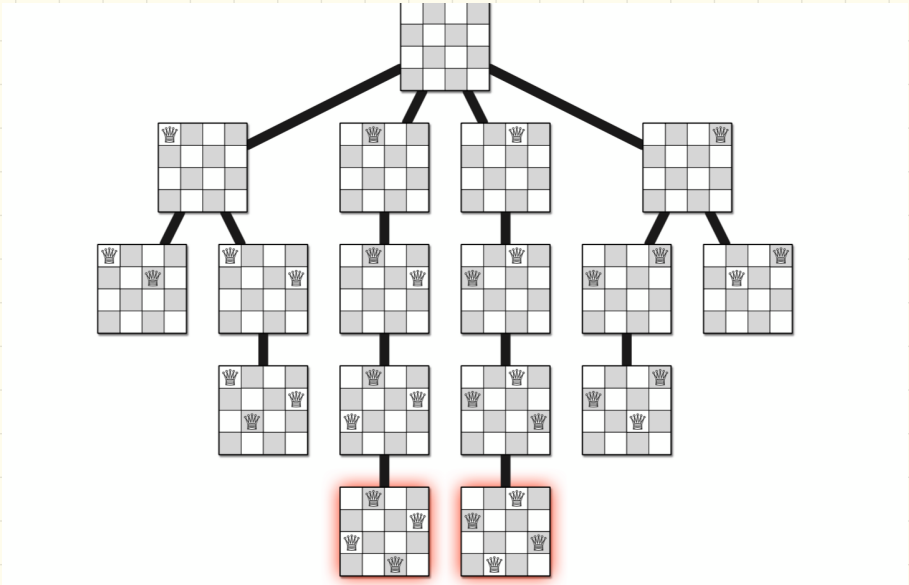

Figure 2.3. The complete recursion tree of Gauss and Laquière's algorithm for the 4 queens problem.

Second:

Simple game

Essentially a variant of Nim:

Nim

For other uses, see Nim (disambiguation).

Nim is a mathematical game of strategy in which two players take turns removing (i.e., nimming) objects from distinct heaps or piles. On each turn, a player must remove at least one object, and may remove any number of objects provided they all come from the same heap/pile. Depending on the version being played, the goal of the game is either to avoid taking the last object, or to take the last object.

Variants of Nim have been played since ancient times.[1] The game is said to have originated in China—it closely resembles the Chinese game of 捡石子 *jiǎn-shízi*, or "picking stones"[2]—but the origin is uncertain; the earliest European references to Nim are from the beginning of the 16th century. Its current name was coined by Charles L. Bouton of Harvard University, who also developed the complete theory of the game in 1901,[3] but the origins of the name were never fully explained.

Key:

Play out success/failure based on game tree.

One note:

These are not greedy!

Greedy would place if it could + keep going.

This one tries placing, + tries <u>not</u> placing.

So: guaranteed to work.

(More formal proofs coming...)

Example: Subset Sum

Given a set X of positive integers and a target value t, is there a subset of X which sums to t?

Ex: $X = \{8, 6, 7, 3, 10, 5, 9\}$

$t = 15$

could include 8
└ t is now 7

not - t is still 15

Now 6:

How would we solve?
(recursion)

Consider recursively:

$$X = \{8, 6, 7, 5, 3, 1, 9\}$$

Formalize this: recursion!

$$T(X[1..n], t) = \begin{cases} \text{include } X[1]: & T(X[2..n], t - X[1]) \\ \text{not}: & T(X[2..n], t) \end{cases}$$

or base case?

$T(\text{empty list}, t>0) = \text{false}$

$T(X[1..n], t<0) = \text{false}$

if $t=0$, true

Pseudocode:

```
SUBSETSUM(X[1..n], T):
  if T = 0
      return TRUE
  else if T < 0 or n = 0
      return FALSE
  else
      return (SUBSETSUM(X[1..n−1], T) ∨ SUBSETSUM(X[1..n−1], T − X[n]))
```

Runtime: Let $S(n)$ be runtime of n element list

$= S(n-1) + S(n-1) + O(1)$

$= 2S(n-1) + O(1)$

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

Correctness:

We are trying every possible subset.

IS: Well, X[n] is either is subset or not.

I recurse on both possibilities & by IH solve them correctly.

Text Segmentation:

Idea: Given a string of "letters", break into "words".

Assume: Given IsWord(w), which takes a string & says true or false.

O(1) time

Back tracking:

Starting at beginning, Check every prefix:

if IsWord(A[1]), recurse on A[2..n]

if IsWord(A[1,2]), try A[3..n]

if IsWord(A[1,2,3]), try A[4..n]

⋮

if IsWord(A[1..i]), try A[i+1,n]

⋮

if IsWord(A[1..n]), done

→ If any succeed, return true

Better pseudocode:

```
SPLITTABLE(A[1..n]):
  if n = 0
      return TRUE
  for i ← 1 to n
      if ISWORD(A[1..i])
          if SPLITTABLE(A[i+1..n])
              return TRUE
  return FALSE
```

Pf of correctness:

We try everything! Why?

Well, A[i] must be in some word, & I try all possible prefixes. By IH, alg checks the rest correctly.

He then gives an index formulation.

I have mixed feeling:

- worth discussing
- BUT - really depends on the language!

Options:

depend on reference, heap, ...

Runtime:

```
SPLITTABLE(A[1..n]):
  if n = 0
    return TRUE
  for i ← 1 to n
    if ISWORD(A[1..i])
      if SPLITTABLE(A[i+1..n])
        return TRUE
  return FALSE
```

$O(1)$

$$T(n) = O(1) + \sum_{i=1}^{n-1} T(i)$$

Icky - but a cool trick helps:

$$T(n) = \sum_{i=1}^{n-1} T(i) + 1$$

write n-1 version:

$$T(n-1) = \sum_{i=1}^{n-2} T(i) + 1$$

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

rearrange:

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

$$= O(2^n)$$

Longest Increasing Subsequence or (LIS)

Given: List of integers $A[1..n]$

Goal: Find longest subsequence whose elements are strictly increasing

Formally: $A[1..n]$, Find largest k s.t. $1 \le i_1 < \dots < i_k \le n$ s.t. $A[i_j] < A[i_{j+1}]$ for every j

Example:

$[12, 5, 1, 3, 4, 13, 6, 11, 2, 20]$

(positions: 3, 4, 5, 7, 8, 10)

IS: 12, 13, 20

Best? length 6 $k=6$ index

Formalize (a la backtracking):

The LIS of $A[1..n]$ is either:

- the LIS of $A[2..n]$
- $A[1]$ followed by LIS of $A[2..n]$

(or is it?)

Go back to that example...

Let:

LISBIGGER(i, j) :=

Then: backtracking recurrence

$$\text{LISBIGGER}(i,j) = \begin{cases} \text{include } j: \\ \\ \text{skip } j: \end{cases} \quad (i < j)$$