Backtracking

Lecture 12

Recursion types

1. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems.

   Examples: Merge sort, quick sort, multiplication, median selection.

2. **Backtracking**

Part I

N Queens Problem

N Queens Problem

Definition

Place n queens on an $n \times n$ board so that no two queens are attacking each other.

N Queens Problem

Definition

Place n queens on an $n \times n$ board so that no two queens are attacking each other.

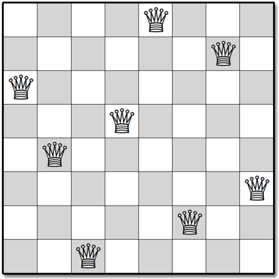
that is, no two queens are in the same row, same column, or same diagonal.

N Queens Problem

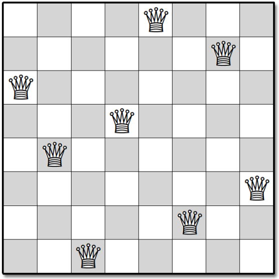
Definition

Place n queens on an $n \times n$ board so that no two queens are attacking each other.

that is, no two queens are in the same row, same column, or same diagonal.



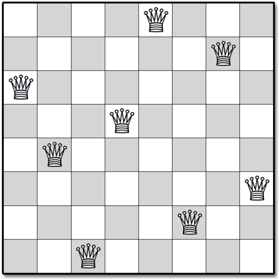
N Queens Problem



Brute-force algorithm:

Try all combinations of n positions.

N Queens Problem



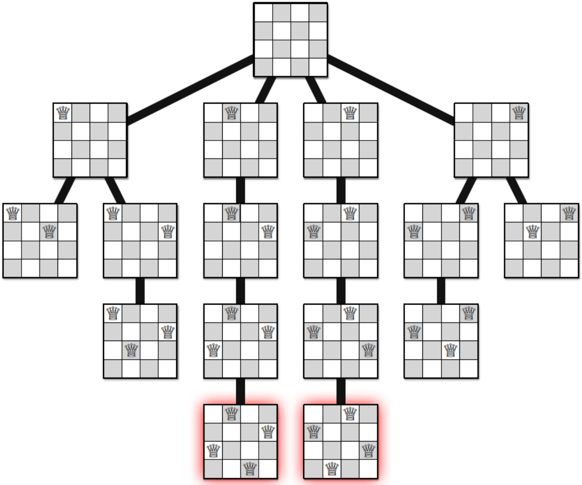
Brute-force algorithm:

Try all combinations of n positions.

Methodical brute-force:

No two queens on the same row, so place a queen in one row at a time.

N Queens Problem



N Queens Problem

Base case

- when any position in the row is attacked by a queen on an earlier row, recursion terminates.
- Or when all n queens are placed.

N Queens Problem

Base case

- when any position in the row is attacked by a queen on an earlier row, recursion terminates.
- Or when all n queens are placed.

Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision
- Let the recursion fairy make all remaining decisions

N Queens Problem

How do we redefine the problem to make recursion work?

N Queens Problem

How do we redefine the problem to make recursion work?

- The recursion does not solve the $n - 1$ queens problem

N Queens Problem

How do we redefine the problem to make recursion work?

- The recursion does not solve the $n - 1$ queens problem
- We need to place the r-th queen so that it is not attacked by a queen on an earlier row

N Queens Problem

How do we redefine the problem to make recursion work?

- The recursion does not solve the $n - 1$ queens problem
- We need to place the r-th queen so that it is not attacked by a queen on an earlier row
- The recursive subproblem:
  - Input $= r - 1$ queens placed in earlier rows
  - Place the remaining $n - r + 1$ queens, one on each row
  - Recurse by increasing r

N Queens Problem

```
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!⟩⟩
```

Recursion types

1. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems.

   Examples: Merge sort, quick sort, multiplication, median selection.

   Each sub-problem is a fraction smaller.

Recursion types

1. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems.

   Examples: Merge sort, quick sort, multiplication, median selection.

   Each sub-problem is a fraction smaller.

2. **Backtracking**: A sequence of decision problems. Recursion tries all possibilities at each step.

   Each subproblem is only a constant smaller, e.g. from n to $n - 1$.

Recursion types

1. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems.

   Examples: Merge sort, quick sort, multiplication, median selection.

   Each sub-problem is a fraction smaller.

2. **Backtracking**: A sequence of decision problems. Recursion tries all possibilities at each step.

   Each subproblem is only a constant smaller, e.g. from n to $n-1$.
   e.g. $T(n) = n \cdot T(n-1)$, $T(1) = n$, hence $T(n) = O(n^n)$.
   e.g. $T(n) = 2 \cdot T(n-1) + O(1)$, hence $T(n) = O(2^n)$.

Part II

Text Segmentation

Problem

Input: A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function **IsStrInL**$(string\ \ x)$ that decides whether x is in L

Goal: Decide if $w \in L^*$ using **IsStrInL**$(string\ \ x)$ as a black box sub-routine

Problem

Input A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function **IsStrInL**$(string\ \ x)$ that decides whether x is in L

Goal Decide if $w \in L^*$ using **IsStrInL**$(string\ \ x)$ as a black box sub-routine

Example

Suppose L is *English* and we have a procedure to check whether a string/word is in the *English* dictionary.

- Is the string "isthisanenglishsentence" in *English**?
- Is "stampstamp" in *English**?
- Is "zibzzzad" in *English**?

Text Segmentation

Backtracking

- Changes the problem into a sequence of decision problems

Text Segmentation

Backtracking

- Changes the problem into a sequence of decision problems

BLUE	**STEM**	**UNIT**	**ROBOT**	HEARTHANDSATURNSPIN

Text Segmentation

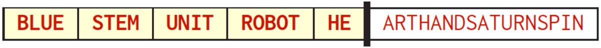
Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision

Text Segmentation

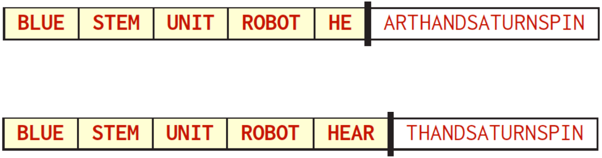
Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision



Backtracking

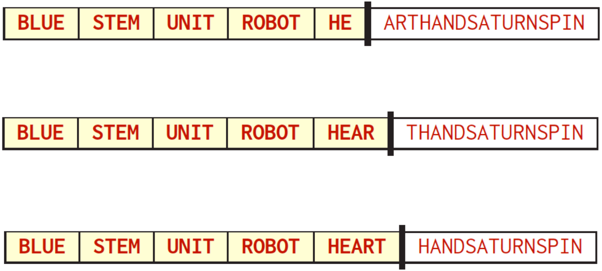
- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision



Text Segmentation

Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision



Text Segmentation

Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN
BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN
BLUE	STEM	UNIT	ROBOT	HEART	HANDSATURNSPIN
BLUE	STEM	UNIT	ROBOT	HEARTH	ANDSATURNSPIN

Text Segmentation

Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision
- Let the recursion fairy make all remaining decisions

Text segmentation

Only the suffix matters.

Text segmentation

Only the suffix matters.

Base case

- zero-length string

Recursive Solution

Assume w is stored in array $A[1..n]$

```
IsStringinLstar(A[1..n]):
    If (n = 0) Output YES
    If (IsStrInL(A[1..n]))
        Output YES
    Else
        For (i = 1 to n − 1) do
            If (IsStrInL(A[1..i]) and IsStrInLstar(A[i + 1..n]))
                Output YES

    Output NO
```

Part III

Longest Increasing Subsequence

Sequences

Definition

Sequence: an ordered list $a_1, a_2, \ldots, a_n$. **Length** of a sequence is number of elements in the list.

Sequences

Definition

Sequence: an ordered list $a_1, a_2, \ldots, a_n$. **Length** of a sequence is number of elements in the list.

Definition

$a_{i_1}, \ldots, a_{i_k}$ is a **subsequence** of $a_1, \ldots, a_n$ if $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

Sequences

Definition

Sequence: an ordered list $a_1, a_2, \ldots, a_n$. **Length** of a sequence is number of elements in the list.

Definition

$a_{i_1}, \ldots, a_{i_k}$ is a **subsequence** of $a_1, \ldots, a_n$ if
$1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

Definition

A sequence is **increasing** if $a_1 < a_2 < \ldots < a_n$. It is **non-decreasing** if $a_1 \leq a_2 \leq \ldots \leq a_n$. Similarly **decreasing** and **non-increasing**.

Sequences

Example...

Example

1. Sequence: $6, 3, 5, 2, 7, 8, 1, 9$
2. Subsequence of above sequence: $5, 2, 1$

Sequences

Example...

Example

1. Sequence: $6, 3, 5, 2, 7, 8, 1, 9$
2. Subsequence of above sequence: $5, 2, 1$
3. Increasing sequence: $3, 5, 9, 17, 54$
4. Decreasing sequence: $34, 21, 7, 5, 1$

Sequences

Example...

Example

1. Sequence: $6, 3, 5, 2, 7, 8, 1, 9$
2. Subsequence of above sequence: $5, 2, 1$
3. Increasing sequence: $3, 5, 9, 17, 54$
4. Decreasing sequence: $34, 21, 7, 5, 1$
5. Increasing subsequence of the first sequence: $2, 7, 9$.

Longest Increasing Subsequence Problem

Input A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal Find an **increasing subsequence** $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

Longest Increasing Subsequence Problem

Input A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal Find an **increasing subsequence** $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
3. Longest increasing subsequence: 3, 5, 7, 8

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($A[1..n]$):

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($A[1..n]$):

1. Case 1: max without $A[n]$ which is LIS($A[1..(n-1)]$)

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($A[1..n]$):

1. Case 1: max without $A[n]$ which is LIS$(A[1..(n-1)])$
2. Case 2: max among sequences that contain $A[n]$ in which case recursion is

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($A[1..n]$):

1. Case 1: max without $A[n]$ which is LIS$(A[1..(n-1)])$
2. Case 2: max among sequences that contain $A[n]$ in which case recursion is not so clear.

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($A[1..n]$):

1. Case 1: max without $A[n]$ which is LIS$(A[1..(n-1)])$
2. Case 2: max among sequences that contain $A[n]$ in which case recursion is not so clear.

Observation

For second case we want to find a subsequence in $A[1..(n-1)]$ *that is restricted to numbers less than* $A[n]$.

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($A[1..n]$):

1. Case 1: max without $A[n]$ which is LIS($A[1..(n-1)]$)
2. Case 2: max among sequences that contain $A[n]$ in which case recursion is not so clear.

Observation

For second case we want to find a subsequence in $A[1..(n-1)]$ that is restricted to numbers less than $A[n]$. This suggests that a more general problem is **LIS_smaller**($A[1..n], x$) *which gives the longest increasing subsequence in A where each number in the sequence is less than x.*

Recursive Approach

LIS_smaller$(A[1..n], x)$ **:** length of longest increasing subsequence in $A[1..n]$ with all numbers in subsequence less than x

```
LIS_smaller(A[1..n], x):
    if (n = 0) then return 0
    m = LIS_smaller(A[1..(n − 1)], x)
    if (A[n] < x) then
        m = max(m, 1 + LIS_smaller(A[1..(n − 1)], A[n]))
    Output m
```

```
LIS(A[1..n]):
        return LIS_smaller(A[1..n], ∞)
```

Part IV

From Backtracking to DP

Running time analysis of Text Segmentation

```
IsStringinLstar(A[1..n]):
    If (n = 0) Output YES
    If (IsStrInL(A[1..n]))
        Output YES
    Else
        For (i = 1 to n - 1) do
            If (IsStrInL(A[1..i]) and IsStrInLstar(A[i + 1..n]))
                Output YES

    Output NO
```

Running time analysis of Text Segmentation

```
IsStringinLstar(A[1..n]):
    If (n = 0) Output YES
    If (IsStrInL(A[1..n]))
        Output YES
    Else
        For (i = 1 to n − 1) do
            If (IsStrInL(A[1..i]) and IsStrInLstar(A[i + 1..n]))
                Output YES

    Output NO
```

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

Running time: $O(2^n)$

Running time analysis of Text Segmentation

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

Running time: $O(2^n)$

However, how many suffixes are there?

Running time analysis of Text Segmentation

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

Running time: $O(2^n)$

However, how many suffixes are there? $O(n)$

Running time analysis of Text Segmentation

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

Running time: $O(2^n)$

HEARTHANDSATURNSPIN

However, how many suffixes are there? $O(n)$

Different past decision can lead to the same suffix.

BLUE	**STEM**	**UNIT**	**ROBOT**	HEARTHANDSATURNSPIN
BLUEST	**EMU**	**NITRO**	**BOT**	HEARTHANDSATURNSPIN