Backtracking and Memoization

Lecture 12
Tuesday, February 26, 2019
Recursion

Reduction:
Reduce one problem to another

Recursion
A special case of reduction
1. reduce problem to a smaller instance of itself
2. self-reduction

1. Problem instance of size $n$ is reduced to one or more instances of size $n - 1$ or less.
2. For termination, problem instances of small size are solved by some other method as base cases.
1. **Tail Recursion**: problem reduced to a *single* recursive call after some work. Easy to convert algorithm into iterative or greedy algorithms. Examples: Interval scheduling, MST algorithms, etc.

2. **Divide and Conquer**: Problem reduced to multiple independent sub-problems that are solved separately. Conquer step puts together solution for bigger problem. Examples: Closest pair, deterministic median selection, quick sort.

3. **Backtracking**: Refinement of brute force search. Build solution incrementally by invoking recursion to try all possibilities for the decision in each step.

4. **Dynamic Programming**: problem reduced to multiple (typically) dependent or overlapping sub-problems. Use memoization to avoid recomputation of common solutions leading to iterative bottom-up algorithm.
Part I

Brute Force Search, Recursion and Backtracking
Maximum Independent Set in a Graph

**Definition**

Given undirected graph $G = (V, E)$ a subset of nodes $S \subseteq V$ is an independent set (also called a stable set) if for there are no edges between nodes in $S$. That is, if $u, v \in S$ then $(u, v) \not\in E$.

Some independent sets in graph above: $\{D\}, \{A, C\}, \{B, E, F\}$
Maximum Independent Set Problem

Input  Graph \( G = (V, E) \)

Goal  Find maximum sized independent set in \( G \)
Maximum Weight Independent Set Problem

Input: Graph $G = (V, E)$, weights $w(v) \geq 0$ for $v \in V$

Goal: Find maximum weight independent set in $G$
No one knows an *efficient* (polynomial time) algorithm for this problem.

Problem is **NP-Complete** and it is *believed* that there is no polynomial time algorithm.

**Brute-force algorithm:**
Try all subsets of vertices.
Brute-force enumeration

Algorithm to find the size of the maximum weight independent set.

\[
\text{MaxIndSet}(G = (V, E)):
\]

\[
\begin{align*}
\text{max} &= 0 \\
\text{for each subset } S \subseteq V \text{ do} \\
&\quad \text{check if } S \text{ is an independent set} \\
&\quad \text{if } S \text{ is an independent set and } w(S) > \text{max} \text{ then} \\
&\quad \hspace{1cm} \text{max} = w(S)
\end{align*}
\]

Output \( \text{max} \)

Running time: suppose \( G \) has \( n \) vertices and \( m \) edges

1. \( 2^n \) subsets of \( V \)
2. checking each subset \( S \) takes \( O(m) \) time
3. total time is \( O(m2^n) \)
Brute-force enumeration

Algorithm to find the size of the maximum weight independent set.

\[
\text{MaxIndSet}(G = (V, E)):
\]
\[
\begin{align*}
\text{max} & = 0 \\
\text{for each subset } S & \subseteq V \text{ do} \\
\text{check if } S & \text{ is an independent set} \\
\text{if } S & \text{ is an independent set and } w(S) > \text{max} \text{ then} \\
\text{max} & = w(S)
\end{align*}
\]

Output \text{max}

Running time: suppose $G$ has $n$ vertices and $m$ edges

1. $2^n$ subsets of $V$
2. checking each subset $S$ takes $O(m)$ time
3. total time is $O(m2^n)$
A Recursive Algorithm

Let \( V = \{v_1, v_2, \ldots, v_n\} \).
For a vertex \( u \) let \( N(u) \) be its neighbors.

**Observation**

\( v_1 \): vertex in the graph.

One of the following two cases is true

- Case 1 \( v_1 \) is in some maximum independent set.
- Case 2 \( v_1 \) is in no maximum independent set.

We can try both cases to “reduce” the size of the problem

\[ G_1 = G - v_1 \] obtained by removing \( v_1 \) and incident edges from \( G \)

\[ G_2 = G - v_1 - N(v_1) \] obtained by removing \( N(v_1) \cup v_1 \) from \( G \)

\[ MIS(G) = \max\{MIS(G_1), MIS(G_2) + w(v_1)\} \]
A Recursive Algorithm

Let $V = \{v_1, v_2, \ldots, v_n\}$.
For a vertex $u$ let $N(u)$ be its neighbors.

**Observation**

$v_1$: vertex in the graph.
One of the following two cases is true

- **Case 1** $v_1$ is in some maximum independent set.
- **Case 2** $v_1$ is in no maximum independent set.

We can try both cases to “reduce” the size of the problem.

$G_1 = G - v_1$ obtained by removing $v_1$ and incident edges from $G$
$G_2 = G - v_1 - N(v_1)$ obtained by removing $N(v_1) \cup v_1$ from $G$

$$\text{MIS}(G) = \max\{\text{MIS}(G_1), \text{MIS}(G_2) + w(v_1)\}$$
A Recursive Algorithm

Let $V = \{v_1, v_2, \ldots, v_n\}$.
For a vertex $u$ let $N(u)$ be its neighbors.

Observation
$v_1$: vertex in the graph.
One of the following two cases is true

Case 1 $v_1$ is in some maximum independent set.
Case 2 $v_1$ is in no maximum independent set.

We can try both cases to “reduce” the size of the problem

$G_1 = G - v_1$ obtained by removing $v_1$ and incident edges from $G$
$G_2 = G - v_1 - N(v_1)$ obtained by removing $N(v_1) \cup v_1$ from $G$

$\text{MIS}(G) = \max\{\text{MIS}(G_1), \text{MIS}(G_2) + w(v_1)\}$
A Recursive Algorithm

RecursiveMIS($G$):

if $G$ is empty then Output 0

$a = \text{RecursiveMIS}(G - v_1)$

$b = w(v_1) + \text{RecursiveMIS}(G - v_1 - N(v_n))$

Output $\max(a, b)$
Recursive Algorithms
..for Maximum Independent Set

Running time:

\[ T(n) = T(n - 1) + T\left(n - 1 - \text{deg}(v_1)\right) + O(1 + \text{deg}(v_1)) \]

where \( \text{deg}(v_1) \) is the degree of \( v_1 \). \( T(0) = T(1) = 1 \) is base case.

Worst case is when \( \text{deg}(v_1) = 0 \) when the recurrence becomes

\[ T(n) = 2T(n - 1) + O(1) \]

Solution to this is \( T(n) = O(2^n) \).
Backtrack Search via Recursion

1. Recursive algorithm generates a tree of computation where each node is a smaller problem (subproblem).

2. Simple recursive algorithm computes/explores the whole tree blindly in some order.

3. Backtrack search is a way to explore the tree intelligently to prune the search space.
   - Some subproblems may be so simple that we can stop the recursive algorithm and solve it directly by some other method.
   - Memoization to avoid recomputing same problem.
   - Stop the recursion at a subproblem if it is clear that there is no need to explore further.
   - Leads to a number of heuristics that are widely used in practice although the worst case running time may still be exponential.
12.1: 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.

**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**.
## 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

### 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
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
Naïve Enumeration

Assume $a_1, a_2, \ldots, a_n$ is contained in an array $A$

```python
algLISNaive(A[1..n]):
    max = 0
    for each subsequence $B$ of $A$ do
        if $B$ is increasing and $|B| > max$ then
            max = $|B|$
    Output max
```

Running time: $O(n2^n)$.

$2^n$ subsequences of a sequence of length $n$ and $O(n)$ time to check if a given sequence is increasing.
Naïve Enumeration

Assume \( a_1, a_2, \ldots, a_n \) is contained in an array \( A \)

\[
\text{algLISNaive}(A[1..n]):
\]
\[
\begin{align*}
\text{max} & = 0 \\
\text{for each subsequence } B \text{ of } A \text{ do} \\
& \quad \text{if } B \text{ is increasing and } |B| > \text{max} \text{ then} \\
& \quad \quad \text{max} = |B|
\end{align*}
\]

Output \( \text{max} \)

Running time: \( O(n2^n) \).

2\(^n\) subsequences of a sequence of length \( n \) and \( O(n) \) time to check if a given sequence is increasing.
Naïve Enumeration

Assume \( a_1, a_2, \ldots, a_n \) is contained in an array \( A \)

\[
\text{algLISNaive}(A[1..n]): \\
\text{max} = 0 \\
\text{for each subsequence } B \text{ of } A \text{ do} \\
\quad \text{if } B \text{ is increasing and } |B| > \text{max} \text{ then} \\
\quad \quad \text{max} = |B| \\
\text{Output } \text{max}
\]

Running time: \( O(n2^n) \).

\( 2^n \) subsequences of a sequence of length \( n \) and \( O(n) \) time to check if a given sequence is increasing.
Recursive Approach: Take 1

**LIS**: Longest increasing subsequence

Can we find a recursive algorithm for **LIS**?

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

1. **Case 1**: Does not contain $A[n]$ in which case
   
   $$\text{LIS}(A[1..n]) = \text{LIS}(A[1..(n - 1)])$$

2. **Case 2**: contains $A[n]$ in which case $\text{LIS}(A[1..n])$ 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: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

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

1. Case 1: Does not contain \( A[n] \) in which case \( \text{LIS}(A[1..n]) = \text{LIS}(A[1..(n - 1)]) \)

2. Case 2: contains \( A[n] \) in which case \( \text{LIS}(A[1..n]) \) 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 \( \text{LIS\_smaller}(A[1..n], x) \) which gives the longest increasing subsequence in \( A \) where each number in the sequence is less than \( x \).
Can we find a recursive algorithm for \textbf{LIS}?

\textbf{LIS}(A[1..n]):

1. Case 1: Does not contain \textbf{A}[n] in which case 
   \textbf{LIS}(A[1..n]) = \textbf{LIS}(A[1..(n - 1)])

2. Case 2: contains \textbf{A}[n] in which case \textbf{LIS}(A[1..n]) is not so clear.

**Observation**

For second case we want to find a subsequence in \textbf{A}[1..(n - 1)] that is restricted to numbers less than \textbf{A}[n]. This suggests that a more general problem is \textbf{LIS}_{\text{smaller}}(A[1..n], x) which gives the longest increasing subsequence in \textbf{A} where each number in the sequence is less than \textbf{x}.
Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

\[ \text{LIS}(A[1..n]): \]

1. Case 1: Does not contain \( A[n] \) in which case
   \[ \text{LIS}(A[1..n]) = \text{LIS}(A[1..(n-1)]) \]
2. Case 2: contains \( A[n] \) in which case \( \text{LIS}(A[1..n]) \) 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 \( \text{LIS\_smaller}(A[1..n], x) \) which gives the longest increasing subsequence in \( A \) where each number in the sequence is less than \( x \).
Recursive Approach

\textbf{LIS\_smaller}(A[1..n], x) : length of longest increasing subsequence in A[1..n] with all numbers in subsequence less than x

\begin{verbatim}
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
\end{verbatim}

\textbf{LIS}(A[1..n]):

\begin{itemize}
    \item return LIS\_smaller(A[1..n], \infty)
\end{itemize}
Example

Sequence:  \( A[1..7] = 6, 3, 5, 2, 7, 8, 1 \)