Backtracking and Memoization

Lecture 10

September 29, 2015

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 $\mathbf{n}$ is reduced to one or more instances of size $\mathbf{n-1}$ or less.
2. For termination, problem instances of small size are solved by some other method as **base cases**.

Recursion in Algorithm Design

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.

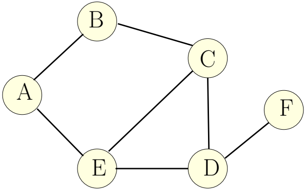
Part I

Brute Force Search, Recursion and Backtracking

Maximum Independent Set in a Graph

Definition

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

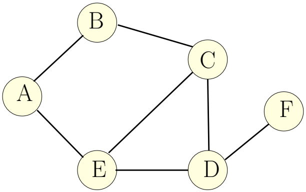


Some independent sets in graph above: $\mathbf{\{D\}, \{A, C\}, \{B, E, F\}}$

Maximum Independent Set Problem

Input Graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$

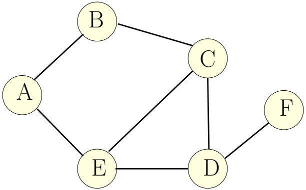
Goal Find maximum sized independent set in $\mathbf{G}$



Maximum Weight Independent Set Problem

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

Goal Find maximum weight independent set in $\mathbf{G}$



Maximum Weight Independent Set Problem

1. No one knows an *efficient* (polynomial time) algorithm for this problem
2. 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.

```
MaxIndSet(G = (V, E)):
    max = 0
    for each subset S ⊆ V do
        check if S is an independent set
        if S is an independent set and w(S) > max then
            max = w(S)

    Output max
```

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

MIS(G)
for each $v \in V$
m_v = MIS(G − v − N(v))

Brute-force enumeration

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

```
MaxIndSet(G = (V, E)):
    max = 0
    for each subset S ⊆ V do
        check if S is an independent set
        if S is an independent set and w(S) > max then
            max = w(S)

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

A Recursive Algorithm

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

Observation

$\mathbf{v_1}$*: vertex in the graph.*
One of the following two cases is true

Case 1 $\mathbf{v_1}$ *is in* some *maximum independent set.*

Case 2 $\mathbf{v_1}$ *is in* no *maximum independent set.*

We can try both cases *to "reduce" the size of the problem*

A Recursive Algorithm

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

Observation

$\mathbf{v_1}$: *vertex in the graph.*
One of the following two cases is true

Case 1 $\mathbf{v_1}$ *is in* some *maximum independent set.*

Case 2 $\mathbf{v_1}$ *is in* no *maximum independent set.*

We can try both cases *to "reduce" the size of the problem*

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

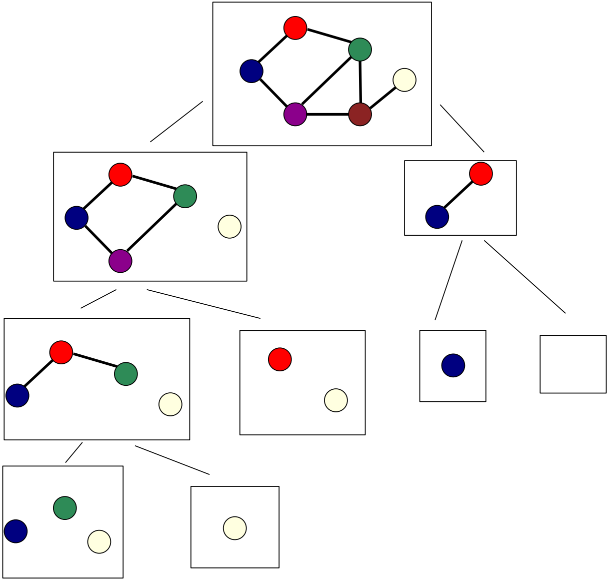
$$\mathbf{MIS(G) = \max\{MIS(G_1), MIS(G_2) + w(v_1)\}}$$

A Recursive Algorithm

```
RecursiveMIS(G):
    if G is empty then Output 0
    a = RecursiveMIS(G − v1)
    b = w(v1) + RecursiveMIS(G − v1 − N(v1))
    Output max(a, b)
```

$$T(n) \leq T(n-1) + T(n-1) + O(n)$$

Example



Recursive Algorithms

..for Maximum Independent Set

Running time:

$$\mathbf{T(n) = T(n-1) + T\Big(n - 1 - deg(v_1)\Big) + O(1 + deg(v_1))}$$

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

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

$$\mathbf{T(n) = 2T(n-1) + O(1)}$$

Solution to this is $\mathbf{T(n) = O(2^n)}$.

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

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: $\mathbf{6, 3, 5, 2, 7, 8, 1, 9}$
2. Subsequence of above sequence: $\mathbf{5, 2, 1}$
3. Increasing sequence: $\mathbf{3, 5, 9, 17, 54}$
4. Decreasing sequence: $\mathbf{34, 21, 7, 5, 1}$
5. Increasing <u>subsequence</u> of the first sequence: $\mathbf{2, 7, 9}$.

Longest Increasing Subsequence Problem

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

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

Longest Increasing Subsequence Problem

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

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

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1, 9
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

```
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
```

Naïve Enumeration

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

```
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:

Naïve Enumeration

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

```
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.

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: Does not contain **A[n]** in which case $\mathrm{LIS}(\mathbf{A[1..n]}) = \mathrm{LIS}(\mathbf{A[1..(n-1)]})$
2. Case 2: contains **A[n]** in which case LIS(**A[1..n]**) is

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 $\mathrm{LIS}(\mathbf{A[1..n]}) = \mathrm{LIS}(\mathbf{A[1..(n-1)]})$
2. Case 2: contains **A[n]** in which case LIS(**A[1..n]**) is not so clear.

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

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

1. Case 1: Does not contain $\mathbf{A[n]}$ in which case $\text{LIS}(\mathbf{A[1..n]}) = \text{LIS}(\mathbf{A[1..(n-1)]})$
2. Case 2: contains $\mathbf{A[n]}$ in which case $\text{LIS}(\mathbf{A[1..n]})$ is not so clear.

Observation

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

Recursive Approach

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

```
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], ∞)
```

Example

Sequence: $\mathbf{A[1..7] = 6, 3, 5, 2, 7, 8, 1}$

Part II

Recursion and Memoization

Fibonacci Numbers

Fibonacci numbers defined by recurrence:

$$\mathbf{F(n) = F(n-1) + F(n-2)} \text{ and } \mathbf{F(0) = 0, F(1) = 1.}$$

These numbers have many interesting and amazing properties.
A journal *The Fibonacci Quarterly*!

1. $\mathbf{F(n) = (\phi^n - (1-\phi)^n)/\sqrt{5}}$ where ϕ is the golden ratio $\mathbf{(1+\sqrt{5})/2 \simeq 1.618}$.
2. $\lim_{n\to\infty} \mathbf{F(n+1)/F(n)} = \phi$

How many bits?

Consider the nth Fibonacci number $F(n)$. Writing the number $F(n)$ in base 2 requires

(A) $\Theta(n^2)$ bits.

(B) $\Theta(n)$ bits.

(C) $\Theta(\log n)$ bits.

(D) $\Theta(\log \log n)$ bits.

Recursive Algorithm for Fibonacci Numbers

Question: Given **n**, compute **F(n)**.

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

$$T(n) = T(n-1) + T(n-2) + O(1)$$
$$\leq T(n-1) + T(n-1) + O(1)$$
$$\leq 2T(n-1) + O(1)$$

Recursive Algorithm for Fibonacci Numbers

Question: Given $\mathbf{n}$, compute $\mathbf{F(n)}$.

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

Running time? Let $\mathbf{T(n)}$ be the number of additions in Fib(n).

Recursive Algorithm for Fibonacci Numbers

Question: Given $\mathbf{n}$, compute $\mathbf{F(n)}$.

```
Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        return Fib(n − 1) + Fib(n − 2)
```

Running time? Let $\mathbf{T(n)}$ be the number of additions in Fib(n).

$$\mathbf{T(n) = T(n-1) + T(n-2) + 1} \text{ and } \mathbf{T(0) = T(1) = 0}$$

Recursive Algorithm for Fibonacci Numbers

Question: Given $\mathbf{n}$, compute $\mathbf{F(n)}$.

```
Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        return Fib(n − 1) + Fib(n − 2)
```

Running time? Let $\mathbf{T(n)}$ be the number of additions in Fib(n).

$$\mathbf{T(n) = T(n-1) + T(n-2) + 1} \text{ and } \mathbf{T(0) = T(1) = 0}$$

Roughly same as $\mathbf{F(n)}$

$$\mathbf{T(n) = \Theta(\phi^n)}$$

The number of additions is exponential in $\mathbf{n}$. Can we do better?

An iterative algorithm for Fibonacci numbers

```
FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    F[0] = 0
    F[1] = 1
    for i = 2 to n do
        F[i] = F[i − 1] + F[i − 2]
    return F[n]
```

An iterative algorithm for Fibonacci numbers

```
FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    F[0] = 0
    F[1] = 1
    for i = 2 to n do
        F[i] = F[i − 1] + F[i − 2]
    return F[n]
```

What is the running time of the algorithm?

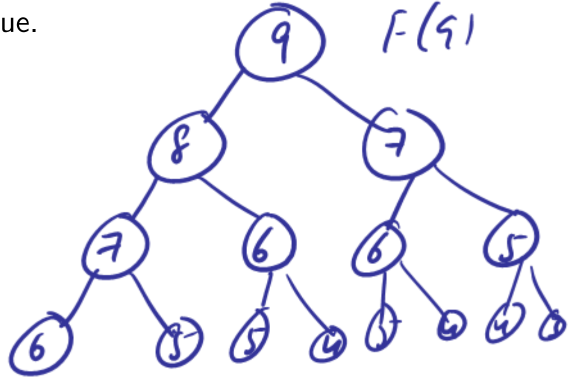
An iterative algorithm for Fibonacci numbers

```
FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    F[0] = 0
    F[1] = 1
    for i = 2 to n do
        F[i] = F[i − 1] + F[i − 2]
    return F[n]
```

What is the running time of the algorithm? $\mathbf{O(n)}$ additions.

What is the difference?

1. Recursive algorithm is computing the same numbers again and again.
2. Iterative algorithm is storing computed values and building bottom up the final value.



What is the difference?

1. Recursive algorithm is computing the same numbers again and again.
2. Iterative algorithm is storing computed values and building bottom up the final value. Memoization.

What is the difference?

1. Recursive algorithm is computing the same numbers again and again.
2. Iterative algorithm is storing computed values and building bottom up the final value. Memoization.

Dynamic Programming:

Fnding a recursion that can be *effectively/efficiently* memoized.

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (Fib(n) was previously computed)
        return stored value of Fib(n)
    else
        return Fib(n − 1) + Fib(n − 2)
```

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (Fib(n) was previously computed)
        return stored value of Fib(n)
    else
        return Fib(n − 1) + Fib(n − 2)
```

How do we keep track of previously computed values?

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (Fib(n) was previously computed)
        return stored value of Fib(n)
    else
        return Fib(n − 1) + Fib(n − 2)
```

How do we keep track of previously computed values?
Two methods: explicitly and implicitly (via data structure)

Automatic explicit memoization

Initialize table/array $\mathbf{M}$ of size $\mathbf{n}$ such that $\mathbf{M[i] = -1}$ for $\mathbf{i = 0, \ldots, n}$.

Automatic explicit memoization

Initialize table/array $\mathbf{M}$ of size $\mathbf{n}$ such that $\mathbf{M[i] = -1}$ for $\mathbf{i = 0, \ldots, n}$.

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (M[n] ≠ −1) (* M[n] has stored value of Fib(n) *)
        return M[n]
    M[n] ⇐ Fib(n − 1) + Fib(n − 2)
    return M[n]
```

To allocate memory need to know upfront the number of subproblems for a given input size $\mathbf{n}$

Automatic implicit memoization

Initialize a (dynamic) dictionary data structure **D** to empty

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (n is already in D)
        return value stored with n in D
        val ⇐ Fib(n − 1) + Fib(n − 2)
    Store (n, val) in D
    return val
```

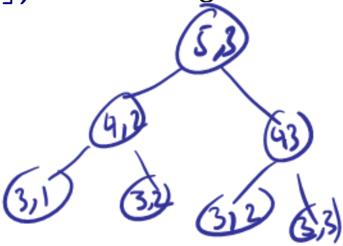
Explicit vs Implicit Memoization

1. Explicit memoization or iterative algorithm preferred if one can analyze problem ahead of time. Allows for efficient memory allocation and access.
2. Implicit and automatic memoization used when problem structure or algorithm is either not well understood or in fact unknown to the underlying system.
    1. Need to pay overhead of data-structure.
    2. Functional languages such as LISP automatically do memoization, usually via hashing based dictionaries.

How many distinct calls?

```
binom(t, b)    // computes (t choose b)
    if t = 0 then return 0
    if b = t or b = 0 then return 1
    return binom(t − 1, b − 1) + binom(t − 1, b).
```

How many distinct calls does **binom(n, $\lfloor n/2 \rfloor$)** makes during its recursive execution?



(A) $\Theta(1)$.

(B) $\Theta(n)$.

(C) $\Theta(n \log n)$.

(D) $\Theta(n^2)$.

(E) $\Theta\left(\binom{n}{\lfloor n/2 \rfloor}\right)$.

That is, if the algorithm calls recursively **binom(17, 5)** about 5000 times during the computation, we count this is a single distinct call.

Running time of memoized binom?

```
D: Initially an empty dictionary.
binomM(t, b)    // computes (t choose b)
    if b = t then return 1
    if b = 0 then return 0
    if D[t, b] is defined then return D[t, b]
    D[t, b] ⇐ binomM(t − 1, b − 1) + binomM(t − 1, b).
    return D[t, b]
```

Assuming that every arithmetic operation takes $O(1)$ time, What is the running time of $\mathrm{binomM}(n, \lfloor n/2 \rfloor)$?

(A) $\Theta(1)$.

(B) $\Theta(n)$.

(C) $\Theta(n^2)$.

(D) $\Theta\left(n^3\right)$.

(E) $\Theta\left(\binom{n}{\lfloor n/2 \rfloor}\right)$.

Back to Fibonacci Numbers

Is the iterative algorithm a *polynomial* time algorithm? Does it take **$O(n)$** time?

Back to Fibonacci Numbers

Is the iterative algorithm a *polynomial* time algorithm? Does it take $\mathbf{O(n)}$ time?

1. input is $\mathbf{n}$ and hence input size is $\mathbf{\Theta(\log n)}$
2. output is $\mathbf{F(n)}$ and output size is $\mathbf{\Theta(n)}$. Why?
3. Hence output size is exponential in input size so no polynomial time algorithm possible!
4. Running time of iterative algorithm: $\mathbf{\Theta(n)}$ additions but number sizes are $\mathbf{O(n)}$ bits long! Hence total time is $\mathbf{O(n^2)}$, in fact $\mathbf{\Theta(n^2)}$. Why?

Back to Fibonacci Numbers

Saving space. Do we need an array of **n** numbers? Not really.

```
FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    prev2 = 0
    prev1 = 1
    for i = 2 to n do
        temp = prev1 + prev2
        prev2 = prev1
        prev1 = temp

    return prev1
```