Binary Search, Introduction to Dynamic Programming

Lecture 7
February 8, 2011
Part I

Exponentiation, Binary Search
Exponentiation

**Input**  Two numbers: \( a \) and integer \( n \geq 0 \)

**Goal**  Compute \( a^n \)

Obvious algorithm:

\[
\text{SlowPow}(a,n): \\
x = 1; \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad x = x \times a \\
\text{Output } x
\]

\( \mathcal{O}(n) \) multiplications.
Exponentiation

Input Two numbers: \( a \) and integer \( n \geq 0 \)

Goal Compute \( a^n \)

Obvious algorithm:

\[
\text{SlowPow}(a, n):
\]
\[
x = 1;
\]
\[
\text{for } i = 1 \text{ to } n \text{ do }
\]
\[
x = x \times a
\]
\[
\text{Output } x
\]

\( O(n) \) multiplications.
Fast Exponentiation

Observation: \( a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} - \lfloor n/2 \rfloor} \).

**FastPow**\((a, n)\):
- if \((n = 0)\) return 1
- \(x = \text{FastPow}(a, \lfloor n/2 \rfloor)\)
- \(x = x \times x\)
- if \((n \text{ is odd})\) then
  - \(x = x \times a\)
- return \(x\)

\(T(n)\): number of multiplications for \(n\)

\[ T(n) \leq T(\lfloor n/2 \rfloor) + 2 \]

\(T(n) = \Theta(\log n)\).
**Fast Exponentiation**

**Observation:** \( a^n = a^{\left\lfloor \frac{n}{2} \right\rfloor} a^{\left\lceil \frac{n}{2} \right\rceil} = a^{\left\lfloor \frac{n}{2} \right\rfloor} a^{\left\lfloor \frac{n}{2} \right\rfloor} a^{\left\lceil \frac{n}{2} \right\rceil - \left\lfloor \frac{n}{2} \right\rfloor} \).

**FastPow\((a,n)\):**

\[
\text{if (n = 0) return 1} \\
\text{x = FastPow(a,\left\lfloor n/2 \right\rfloor)} \\
x = x*x \\
\text{if (n is odd) then} \\
x = x * a \\
\text{return } x
\]

**T(n):** number of multiplications for \( n \)

\[
T(n) \leq T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + 2
\]

\( T(n) = \Theta(\log n) \).
Fast Exponentiation

Observation: \( a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor} \).

\textbf{FastPow}(a, n):

if (n = 0) return 1
x = \textbf{FastPow}(a, \lfloor n/2 \rfloor)
x = x*x
if (n is odd) then
x = x * a
return x

\textbf{T}(n): number of multiplications for \( n \)

\[ T(n) \leq T(\lfloor n/2 \rfloor) + 2 \]

\[ T(n) = \Theta(\log n). \]
Observation: \( a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} - \lfloor n/2 \rfloor \).

\textbf{FastPow}(a,n):

\begin{verbatim}
if (n == 0) return 1
x = FastPow(a,\lfloor n/2 \rfloor)
x = x*x
if (n is odd) then
    x = x * a
return x
\end{verbatim}

\( T(n) \): number of multiplications for \( n \)

\[ T(n) \leq T(\lfloor n/2 \rfloor) + 2 \]

\( T(n) = \Theta(\log n) \).
Fast Exponentiation

Observation: \( a^n = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} - \lfloor n/2 \rfloor} \).

\textbf{FastPow}(a,n):

\begin{itemize}
  \item if (n = 0) return 1
  \item x = \textbf{FastPow}(a,\lfloor n/2 \rfloor)
  \item x = x*x
  \item if (n is odd) then
    \item x = x * a
  \item return x
\end{itemize}

\( T(n) \): number of multiplications for \( n \)

\[ T(n) \leq T(\lfloor n/2 \rfloor) + 2 \]

\( T(n) = \Theta(\log n) \).
Complexity of Exponentiation

**Question:** Is SlowPow() a polynomial time algorithm? FastPow?

Input size: $\log a + \log n$
Output size: $n \log a$. Not necessarily polynomial in input size!

Both **FastPow** and **FastPow** are polynomial in output size.
**Question:** Is SlowPow() a polynomial time algorithm? FastPow?

Input size: \( \log a + \log n \)

Output size: \( n \log a \). Not necessarily polynomial in input size!

Both FastPow and FastPow are polynomial in output size.
Question: Is SlowPow() a polynomial time algorithm? FastPow?

Input size: \( \log a + \log n \)
Output size: \( n \log a \). Not necessarily polynomial in input size!

Both FastPow and FastPow are polynomial in output size.
Complexity of Exponentiation

**Question:** Is SlowPow() a polynomial time algorithm? FastPow?

Input size: $\log a + \log n$
Output size: $n \log a$. Not necessarily polynomial in input size!

Both **FastPow** and **FastPow** are polynomial in output size.
Exponentiation modulo a given number

Exponentiation in applications:

Input  Three integers: $a, n \geq 0, p \geq 2$ (typically a prime)
Goal   Compute $a^n \mod p$

Input size: $\Theta(\log a + \log n + \log p)$
Output size: $O(\log p)$ and hence polynomial in input size.

Observation: $xy \mod p = ((x \mod p)(y \mod p)) \mod p$
Exponentiation modulo a given number

Exponentiation in applications:

**Input**  Three integers:  \( a, n \geq 0, p \geq 2 \)  (typically a prime)

**Goal**  Compute \( a^n \mod p \)

Input size:  \( \Theta(\log a + \log n + \log p) \)

Output size:  \( O(\log p) \)  and hence polynomial in input size.

Observation:  \( xy \mod p = ((x \mod p)(y \mod p)) \mod p \)
Exponentiation modulo a given number

Exponentiation in applications:

**Input** Three integers: \( a, n \geq 0, p \geq 2 \) (typically a prime)

**Goal** Compute \( a^n \mod p \)

Input size: \( \Theta(\log a + \log n + \log p) \)
Output size: \( O(\log p) \) and hence polynomial in input size.

**Observation:** \( xy \mod p = ((x \mod p)(y \mod p)) \mod p \)
Exponentiation modulo a given number

Input Three integers: $a, n \geq 0, p \geq 2$ (typically a prime)

Goal Compute $a^n \mod p$

FastPowMod($a,n,p$):
  if ($n = 0$) return 1
  $x = $ FastPowMod($a,\lfloor n/2 \rfloor, p$)
  $x = x \times x \mod p$
  if ($n$ is odd)
    $x = x \times a \mod p$
  return $x$

FastPowMod is a polynomial time algorithm. SlowPowMod is not (why?).
Exponentiation modulo a given number

Input  Three integers: \(a, n \geq 0, p \geq 2\) (typically a prime)

Goal  Compute \(a^n \mod p\)

\[
\text{FastPowMod}(a, n, p):
\]
\[
\text{if (n = 0) return 1}
\]
\[
x = \text{FastPowMod}(a, \left\lfloor \frac{n}{2} \right\rfloor, p)
\]
\[
x = x \times x \mod p
\]
\[
\text{if (n is odd)}
\]
\[
x = x \times a \mod p
\]
\[
\text{return } x
\]

FastPowMod is a polynomial time algorithm. SlowPowMod is not (why?).
Binary Search in Sorted Arrays

**Input** Sorted array $A$ of $n$ numbers and number $x$

**Goal** Is $x$ in $A$?

BinarySearch($A[a..b]$, $x$):
- if ($b-a \leq 0$) return NO
- mid = $A[\lfloor (a+b)/2 \rfloor]$
- if ($x = \text{mid}$) return YES
- else if ($x < \text{mid}$) return BinarySearch($A[a..\lfloor (a+b)/2 \rfloor - 1],x$)
- else return BinarySearch($A[\lfloor (a+b)/2 \rfloor + 1..b],x$)

**Analysis:** $T(n) = T(\lfloor n/2 \rfloor) + O(1)$. $T(n) = O(\log n)$.

**Observation:** After $k$ steps, size of array left is $n/2^k$
Binary Search in Sorted Arrays

Input  Sorted array \( A \) of \( n \) numbers and number \( x \)

Goal  Is \( x \) in \( A \)?

BinarySearch(\( A[a..b] \), \( x \)):
  if \((b-a) \leq 0\) return NO
  mid = \( A[\lfloor (a + b)/2 \rfloor] \)
  if \((x = \text{mid})\) return YES
  else if \((x < \text{mid})\) return BinarySearch(\( A[a..\lfloor (a + b)/2 \rfloor - 1] \), \( x \))
  else return BinarySearch(\( A[\lfloor (a + b)/2 \rfloor + 1..b] \), \( x \))

Analysis: \( T(n) = T(\lfloor n/2 \rfloor) + O(1). \ T(n) = O(\log n) \).

Observation: After \( k \) steps, size of array left is \( n/2^k \).
Binary Search in Sorted Arrays

Input: Sorted array $A$ of $n$ numbers and number $x$

Goal: Is $x$ in $A$?

```
BinarySearch(A[a..b], x):
    if (b-a <= 0) return NO
    mid = A[⌊(a + b)/2⌋]
    if (x = mid) return YES
    else if (x < mid) return BinarySearch(A[a..⌊(a + b)/2⌋−1], x)
    else return BinarySearch(A[⌊(a + b)/2⌋+1..b], x)
```

Analysis: $T(n) = T(⌊n/2⌋) + O(1)$. $T(n) = O(\log n)$.

Observation: After $k$ steps, size of array left is $n/2^k$
Another common use of binary search

- **Optimization version**: find solution of best (say minimum) value
- **Decision version**: is there a solution of value at most a given value \( v \)?

Reduce optimization to decision (may be easier to think about):

- Given instance \( I \) compute upper bound \( U(I) \) on best value
- Compute lower bound \( L(I) \) on best value
- Do binary search on interval \([L(I), U(I)]\) using decision version as black box
- \( O(\log(U(I) - L(I))) \) calls to decision version if \( U(I), L(I) \) are integers
Another common use of binary search

- **Optimization version**: find solution of best (say minimum) value
- **Decision version**: is there a solution of value at most a given value \( v \)?

Reduce optimization to decision (may be easier to think about):

- Given instance \( I \) compute upper bound \( U(I) \) on best value
- Compute lower bound \( L(I) \) on best value
- Do binary search on interval \( [L(I), U(I)] \) using decision version as black box
- \( O(\log(U(I) - L(I))) \) calls to decision version if \( U(I), L(I) \) are integers
Example

- **Problem:** shortest paths in a graph.

- **Decision version:** given $G$ with non-negative integer edge lengths, nodes $s, t$ and bound $B$, is there an $s$-$t$ path in $G$ of length at most $B$?

- **Optimization version:** find the length of a shortest path between $s$ and $t$ in $G$.

**Question:** given a black box algorithm for the decision version, can we obtain an algorithm for the optimization version?
Example continued

**Question:** given a black box algorithm for the decision version, can we obtain an algorithm for the optimization version?

- Let \( U \) be maximum edge length in \( G \).
- Minimum edge length is \( L \).
- \( s-t \) shortest path length is at most \((n - 1)U\) and at least \( L \).
- Apply binary search on the interval \([L, (n - 1)U]\) via the algorithm for the decision problem.
- \( O(\log((n - 1)U - L)) \) calls to the decision problem algorithm sufficient. Polynomial in input size.
Part II

Introduction to Dynamic Programming
Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

- reduce problem to a *smaller* instance of *itself*
- self-reduction

- Problem instance of size $n$ is reduced to one or more instances of size $n - 1$ or less.
- For termination, problem instances of small size are solved by some other method as *base cases*
Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

- reduce problem to a *smaller* instance of *itself*
- self-reduction

- Problem instance of size $n$ is reduced to one or more instances of size $n - 1$ or less.
- For termination, problem instances of small size are solved by some other method as *base cases*
Recursion in Algorithm Design

- **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.

- **Divide and Conquer**: problem reduced to multiple independent sub-problems that are solved separately. Conquer step puts together solution for bigger problem.

- **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.
Fibonacci Numbers

Fibonacci numbers defined by recurrence:

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

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

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

**Question:** Given \( n \), compute \( F(n) \).
Fibonacci Numbers

Fibonacci numbers defined by recurrence:

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

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

- \[ F(n) = \left( \phi^n - (1 - \phi)^n \right) / \sqrt{5} \] where \( \phi \) is the golden ratio \( (1 + \sqrt{5})/2 \approx 1.618. \)
- \[ \lim_{n \to \infty} F(n + 1)/F(n) = \phi \]

**Question:** Given \( n \), compute \( F(n) \).
Recursive Algorithm for Fibonacci Numbers

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

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

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

Roughly same as $F(n)$

$$T(n) = \Theta(\phi^n)$$

The number of additions is exponential in $n$. Can we do better?
Recursive Algorithm for Fibonacci Numbers

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

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

$$T(n) = T(n - 1) + T(n - 2) + 1$$ and $T(0) = T(1) = 0$

Roughly same as $F(n)$

$$T(n) = \Theta(\phi^n)$$

The number of additions is exponential in $n$. Can we do better?
Recursive Algorithm for Fibonacci Numbers

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

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

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

Roughly same as $F(n)$

$$T(n) = \Theta(\phi^n)$$

The number of additions is exponential in $n$. Can we do better?
Recursive Algorithm for Fibonacci Numbers

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

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

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

Roughly same as $F(n)$

$$T(n) = \Theta(\phi^n)$$

The number of additions is exponential in $n$. Can we do better?
An iterative algorithm for Fibonacci numbers

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        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? \( O(n) \) additions.
An iterative algorithm for Fibonacci numbers

Fib(n):
  if (n = 0)
    return 0
  else if (n = 1)
    return 1
  else
    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? $O(n)$ additions.
An iterative algorithm for Fibonacci numbers

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        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? $O(n)$ additions.
What is the difference?

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

Dynamic Programming: finding a recursion that can be effectively/efficiently memoized

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

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

Dynamic Programming: finding a recursion that can be *effectively/efficiently* memoized

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

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

Dynamic Programming: finding 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?

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else 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 Memoization

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

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else 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 Memoization

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

Fib(n):
  if (n = 0)
    return 0
  else if (n = 1)
    return 1
  else 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 Memoization

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

Fib(n):
    if \( n = 0 \)
        return 0
    else if \( n = 1 \)
        return 1
    else 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 $M$ of size $n$ such that $M[i] = -1$ for $0 \leq i < n$

Fib(n):
  if (n = 0)
    return 0
  else if (n = 1)
    return 1
  else if ($M[n] \neq -1$) (* $M[n]$ has stored value of Fib(n) *)
    return $M[n]$
  else
    $M[n] = \text{Fib}(n-1) + \text{Fib}(n-2)$
    return $M[n]$

Need to know upfront the number of subproblems to allocate memory
Automatic explicit memoization

Initialize table/array $M$ of size $n$ such that $M[i] = -1$ for $0 \leq i < n$

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else if (M[n] $\neq -1$) (* M[n] has stored value of Fib(n) *)
        return M[n]
    else
        M[n] = Fib(n-1) + Fib(n-2)
        return M[n]

Need to know upfront the number of subproblems to allocate memory.
Automatic implicit memoization

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

\[
\text{Fib}(n):
\begin{align*}
\text{if (}n\text{ = 0)} & \quad \text{return 0} \\
\text{else if (}n\text{ = 1)} & \quad \text{return 1} \\
\text{else if (}n\text{ is already in } D\text{)} & \quad \text{return value stored with } n \text{ in } D \\
\text{else} & \\
\text{val} & = \text{Fib}(n-1) + \text{Fib}(n-2) \\
\text{Store } (n, \text{val}) & \text{ in } D \\
\text{return } \text{val}
\end{align*}
\]
Explicit vs Implicit Memoization

- Explicit memoization or iterative algorithm preferred if one can analyze problem ahead of time. Allows for efficient memory allocation and access.
- Implicit and automatic memoization used when problem structure or algorithm is either not well understood or in fact unknown to the underlying system
  - need to pay overhead of datastructure
  - Functional languages such as LISP automatically do memoization, usually via hashing based dictionaries.
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- Input is $n$ and hence input size is $\Theta(\log n)$
- Output is $F(n)$ and output size is $\Theta(n)$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!

Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?

Running time of recursive algorithm is $O(n\phi^n)$ but can in fact shown to be $O(\phi^n)$ by being careful. Doubly exponential in input size and exponential even in output size.
Is the iterative algorithm a \textit{polynomial} time algorithm? Does it take \( O(n) \) time?

- input is \( n \) and hence input size is \( \Theta(\log n) \)
- output is \( F(n) \) and output size is \( \Theta(n) \). Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
- Running time of iterative algorithm: \( \Theta(n) \) additions but number sizes are \( O(n) \) bits long! Hence total time is \( O(n^2) \), in fact \( \Theta(n^2) \). Why?
- Running time of recursive algorithm is \( O(n\phi^n) \) but can in fact shown to be \( O(\phi^n) \) by being careful. Doubly exponential in input size and exponential even in output size.
Is the iterative algorithm a polynomial time algorithm? Does it take $O(n)$ time?

- input is $n$ and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$. Why?

Hence output size is exponential in input size so no polynomial time algorithm possible!

- Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?

- Running time of recursive algorithm is $O(n\phi^n)$ but can in fact shown to be $O(\phi^n)$ by being careful. Doubly exponential in input size and exponential even in output size.
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is $n$ and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$. Why?

Hence output size is exponential in input size so no polynomial time algorithm possible!

- Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?

- Running time of recursive algorithm is $O(n\phi^n)$ but can in fact shown to be $O(\phi^n)$ by being careful. Doubly exponential in input size and exponential even in output size.
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is $n$ and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!

Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?

Running time of recursive algorithm is $O(n\phi^n)$ but can in fact shown to be $O(\phi^n)$ by being careful. Doubly exponential in input size and exponential even in output size.
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is $n$ and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$. Why?

Hence output size is exponential in input size so no polynomial time algorithm possible!

Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?

Running time of recursive algorithm is $O(n\phi^n)$ but can in fact shown to be $O(\phi^n)$ by being careful. Doubly exponential in input size and exponential even in output size.
Part III

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:
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 $G$
Maximum Weight Independent Set Problem

- 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

A brute-force algorithm: try all subsets of vertices.
Brute-force enumeration

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

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

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

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

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

\textbf{MaxIndSet}(G = (V, E)):
\begin{itemize}
\item max = 0
\item for each subset \( S \subseteq V \) do
\item \hspace{1em} check if \( S \) is an independent set
\item \hspace{1em} if \( S \) is an independent set and \( w(S) > \text{max} \) then
\item \hspace{2em} max = w(S)
\end{itemize}
Output max

Running time: suppose \( G \) has \( n \) vertices and \( m \) edges
\begin{itemize}
\item \( 2^n \) subsets of \( V \)
\item checking each subset \( S \) takes \( O(m) \) time
\item total time is \( O(m2^n) \)
\end{itemize}
A Recursive Algorithm

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

Observation
$v_n$: Vertex in the graph.
One of the following two cases is true

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

RecursiveMIS($G$):
if $G$ is empty then Output 0
$a = \text{RecursiveMIS}(G - v_n)$
$b = w(v_n) + \text{RecursiveMIS}(G - v_n - N(v_n))$
Output $\max(a, b)$
A Recursive Algorithm

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

Observation

$v_n$: Vertex in the graph.

One of the following two cases is true

Case 1 $v_n$ is in some maximum independent set.

Case 2 $v_n$ is in no maximum independent set.

RecursiveMIS($G$):

if $G$ is empty then Output 0

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

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

Output $\max(a, b)$
A Recursive Algorithm

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

Observation

\( v_n \): Vertex in the graph.

One of the following two cases is true

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

**RecursiveMIS** \((G)\):

if \( G \) is empty then Output 0

\[ a = \text{RecursiveMIS}(G - v_n) \]
\[ b = w(v_n) + \text{RecursiveMIS}(G - v_n - N(v_n)) \]

Output \( \max(a, b) \)
Recursive Algorithms

..for Maximum Independent Set

Running time:

\[ T(n) = T(n - 1) + T\left(n - 1 - \deg(v_n)\right) + O(1 + \deg(v_n)) \]

where \( \deg(v_n) \) is the degree of \( v_n \). \( T(0) = T(1) = 1 \) is base case.

Worst case is when \( \deg(v_n) = 0 \) when the recurrence becomes

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

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

- Recursive algorithm generates a tree of computation where each node is a smaller problem (subproblem).
- Simple recursive algorithm computes/explores the whole tree blindly in some order.
- 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 recursing 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.
Example