Binary Search, Introduction to Dynamic Programming

Lecture 7

February 8, 2011

Part I

Exponentiation, Binary Search

Exponentiation

Input Two numbers: $\mathbf{a}$ and integer $\mathbf{n} \geq \mathbf{0}$

Goal Compute $\mathbf{a^n}$

Obvious algorithm:

```
SlowPow(a,n):
    x = 1;
    for i = 1 to n do
        x = x*a
    Output x
```

$O(n)$ multiplications.

Exponentiation

Input Two numbers: $\mathbf{a}$ and integer $\mathbf{n} \geq \mathbf{0}$

Goal Compute $\mathbf{a^n}$

Obvious algorithm:

```
SlowPow(a,n):
    x = 1;
    for i = 1 to n do
        x = x*a
    Output x
```

$\mathbf{O(n)}$ multiplications.

Fast Exponentiation

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

```
FastPow(a,n):
    if (n = 0) return 1
    x = FastPow(a,⌊n/2⌋)
    x = x*x
    if (n is odd) then
        x = x * 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: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a,n):
    if (n = 0) return 1
    x = FastPow(a,⌊n/2⌋)
    x = x*x
    if (n is odd) then
        x = x * 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: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a,n):
    if (n = 0) return 1
    x = FastPow(a,⌊n/2⌋)
    x = x*x
    if (n is odd) then
        x = x * 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: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a,n):
    if (n = 0) return 1
    x = FastPow(a,⌊n/2⌋)
    x = x*x
    if (n is odd) then
        x = x * 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: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a,n):
    if (n = 0) return 1
    x = FastPow(a,⌊n/2⌋)
    x = x*x
    if (n is odd) then
        x = x * a
    return x
```

$T(n)$: 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: $\mathbf{\log a + \log n}$
Output size: $\mathbf{n \log a}$. 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.

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.

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: $\mathbf{a}$, $\mathbf{n} \geq \mathbf{0}$, $\mathbf{p} \geq \mathbf{2}$ (typically a prime)

Goal Compute $\mathbf{a^n \mod p}$

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: $\mathbf{a}$, $\mathbf{n} \geq \mathbf{0}$, $\mathbf{p} \geq \mathbf{2}$ (typically a prime)

Goal Compute $\mathbf{a}^{\mathbf{n}} \mod \mathbf{p}$

```
FastPowMod(a,n,p):
    if (n = 0) return 1
    x = FastPowMod(a,⌊n/2⌋,p)
    x = x*x  mod  p
    if (n is odd)
        x = x*a  mod p
    return x
```

FastPowMod is a polynomial time algorithm. SlowPowMod is not (why?).

Exponentiation modulo a given number

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

Goal Compute $\mathbf{a^n} \mod \mathbf{p}$

```
FastPowMod(a,n,p):
    if (n = 0) return 1
    x = FastPowMod(a,⌊n/2⌋,p)
    x = x*x  mod  p
    if (n is odd)
        x = x*a  mod p
    return x
```

FastPowMod is a polynomial time algorithm. SlowPowMod is not (why?).

Binary Search in Sorted Arrays

Input Sorted array $\mathbf{A}$ of $\mathbf{n}$ numbers and number $\mathbf{x}$

Goal Is $\mathbf{x}$ in $\mathbf{A}$?

```
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: $\mathbf{T(n) = T(\lfloor n/2 \rfloor) + O(1)}$. $\mathbf{T(n) = O(\log n)}$.

Observation: After $\mathbf{k}$ steps, size of array left is $\mathbf{n/2^k}$

Binary Search in Sorted Arrays

Input Sorted array $\mathbf{A}$ of $\mathbf{n}$ numbers and number $\mathbf{x}$

Goal Is $\mathbf{x}$ in $\mathbf{A}$?

```
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: $\mathbf{T(n) = T(\lfloor n/2 \rfloor) + O(1)}$. $\mathbf{T(n) = O(\log n)}$.

Observation: After $\mathbf{k}$ steps, size of array left is $\mathbf{n/2^k}$

Binary Search in Sorted Arrays

Input Sorted array $\mathbf{A}$ of $\mathbf{n}$ numbers and number $\mathbf{x}$

Goal Is $\mathbf{x}$ in $\mathbf{A}$?

```
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: $\mathbf{T(n) = T(\lfloor n/2 \rfloor) + O(1)}$. $\mathbf{T(n) = O(\log n)}$.

Observation: After $\mathbf{k}$ steps, size of array left is $\mathbf{n/2^k}$

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 $\mathbf{v}$?

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

- Given instance $\mathbf{I}$ compute upper bound $\mathbf{U(I)}$ on best value
- Compute lower bound $\mathbf{L(I)}$ on best value
- Do binary search on interval $[\mathbf{L(I)}, \mathbf{U(I)}]$ using decision version as black box
- $\mathbf{O(\log(U(I) - L(I)))}$ calls to decision version if $\mathbf{U(I)}, \mathbf{L(I)}$ 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 $\mathbf{v}$?

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

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

Example

- Problem: shortest paths in a graph.
- Decision version: given $\mathbf{G}$ with non-negative integer edge lengths, nodes $\mathbf{s}, \mathbf{t}$ and bound $\mathbf{B}$, is there an $\mathbf{s}$-$\mathbf{t}$ path in $\mathbf{G}$ of length at most $\mathbf{B}$?
- Optimization version: find the length of a shortest path between $\mathbf{s}$ and $\mathbf{t}$ in $\mathbf{G}$.

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 $\mathbf{U}$ be maximum edge length in $\mathbf{G}$.
- Minimum edge length is $\mathbf{L}$.
- $\mathbf{s}$-$\mathbf{t}$ shortest path length is at most $(\mathbf{n}-1)\mathbf{U}$ and at least $\mathbf{L}$.
- Apply binary search on the interval $[\mathbf{L}, (\mathbf{n}-1)\mathbf{U}]$ 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 $\mathbf{n}$ is reduced to one or more instances of size $\mathbf{n} - \mathbf{1}$ 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 $\mathbf{n}$ is reduced to one or more instances of size $\mathbf{n-1}$ 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:

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

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

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

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

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*!

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

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

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 $\mathbf{T(n)}$ 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?

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 $\mathbf{T(n)}$ 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?

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 $\mathbf{T(n)}$ 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?

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 $\mathbf{T(n)}$ 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

```
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? $\mathbf{O(n)}$ 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 $\mathbf{M}$ of size $\mathbf{n}$ such that $\mathbf{M[i] = -1}$ for $\mathbf{0 \leq i < n}$

```
Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else if (M[n] ≠ -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 explicit memoization

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

```
Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else if (M[n] ≠ -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

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

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.

Back to Fibonacci Numbers

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

- input is $\mathbf{n}$ and hence input size is $\mathbf{\Theta(\log n)}$
- output is $\mathbf{F(n)}$ and output size is $\mathbf{\Theta(n)}$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
- 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?
- Running time of recursive algorithm is $\mathbf{O(n\phi^n)}$ but can in fact shown to be $\mathbf{O(\phi^n)}$ by being careful. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

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

- input is $\mathbf{n}$ and hence input size is $\mathbf{\Theta(\log n)}$
- output is $\mathbf{F(n)}$ and output size is $\mathbf{\Theta(n)}$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
- 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?
- Running time of recursive algorithm is $\mathbf{O(n\phi^n)}$ but can in fact shown to be $\mathbf{O(\phi^n)}$ by being careful. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

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

- input is $\mathbf{n}$ and hence input size is $\mathbf{\Theta(\log n)}$
- output is $\mathbf{F(n)}$ and output size is $\mathbf{\Theta(n)}$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
- 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?
- Running time of recursive algorithm is $\mathbf{O(n\phi^n)}$ but can in fact shown to be $\mathbf{O(\phi^n)}$ by being careful. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

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

- input is $\mathbf{n}$ and hence input size is $\mathbf{\Theta(\log n)}$
- output is $\mathbf{F(n)}$ and output size is $\mathbf{\Theta(n)}$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
- 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?
- Running time of recursive algorithm is $\mathbf{O(n\phi^n)}$ but can in fact shown to be $\mathbf{O(\phi^n)}$ by being careful. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

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

- input is $\mathbf{n}$ and hence input size is $\mathbf{\Theta(\log n)}$
- output is $\mathbf{F(n)}$ and output size is $\mathbf{\Theta(n)}$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
- 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?
- Running time of recursive algorithm is $\mathbf{O(n\phi^n)}$ but can in fact shown to be $\mathbf{O(\phi^n)}$ by being careful. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

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

- input is $\mathbf{n}$ and hence input size is $\mathbf{\Theta(\log n)}$
- output is $\mathbf{F(n)}$ and output size is $\mathbf{\Theta(n)}$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
- 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?
- Running time of recursive algorithm is $\mathbf{O(n\phi^n)}$ but can in fact shown to be $\mathbf{O(\phi^n)}$ by being careful. Doubly exponential in input size and exponential even in output size.

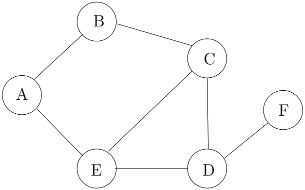
Part III

Brute Force Search, Recursion and Backtracking

Maximum Independent Set in a Graph

Definition

Given undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ a subset of nodes $\mathbf{S} \subseteq \mathbf{V}$ is an independent set (also called a stable set) if for there are no edges between nodes in $\mathbf{S}$. That is, if $\mathbf{u}, \mathbf{v} \in \mathbf{S}$ then $(\mathbf{u}, \mathbf{v}) \notin \mathbf{E}$.

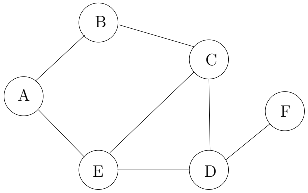


Some independent sets in graph above:

Maximum Independent Set Problem

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

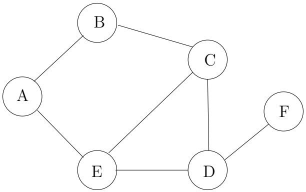
Goal Find maximum sized independent set in $\mathbf{G}$



Maximum Weight Independent Set Problem

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

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



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.

MaxIndSet$(\mathbf{G} = (\mathbf{V}, \mathbf{E}))$:
    $\mathbf{max} = 0$
    for each subset $\mathbf{S} \subseteq \mathbf{V}$ **do**
        check if $\mathbf{S}$ is an independent set
        if $\mathbf{S}$ is an independent set and $\mathbf{w}(\mathbf{S}) > \mathbf{max}$ **then**
            $\mathbf{max} = \mathbf{w}(\mathbf{S})$
    Output $\mathbf{max}$

Running time: suppose $\mathbf{G}$ has $\mathbf{n}$ vertices and $\mathbf{m}$ edges

- $\mathbf{2^n}$ subsets of $\mathbf{V}$
- checking each subset $\mathbf{S}$ takes $\mathbf{O(m)}$ time
- total time is $\mathbf{O(m2^n)}$

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

- 2^n subsets of V
- checking each subset S takes $O(m)$ time
- 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 neighbours.

Observation

$\mathbf{v_n}$: *Vertex in the grpah.*
One of the following two cases is true

- Case 1 $\mathbf{v_n}$ *is in* some *maximum independent set.*
- Case 2 $\mathbf{v_n}$ *is in* no *maximum independent set.*

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

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 neighbours.

Observation

$\mathbf{v_n}$: *Vertex in the grpah.*
One of the following two cases is true

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

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

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

A Recursive Algorithm

Let $\mathbf{V} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$.
For a vertex $\mathbf{u}$ let $\mathbf{N}(\mathbf{u})$ be its neighbours.

Observation

$\mathbf{v}_n$: *Vertex in the grpah.*
One of the following two cases is true

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

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

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

Running time:

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

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

Worst case is when $\mathbf{deg(v_n) = 0}$ 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

- 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 subprobles 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

Notes

Notes

Notes

Notes