CS 473: Fundamental Algorithms, Spring 2013

Binary Search, Introduction to Dynamic Programming

Lecture 7

February 9, 2013

Part I

Exponentiation, Binary Search

Exponentiation

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

Goal Compute a^n

Obvious algorithm:

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

$O(n)$ multiplications.

Fast Exponentiation

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

```
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: $O(\log a + \log n)$
Output size: $O(n \log a)$.

Not necessarily polynomial in input size!

Both **SlowPow** 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

- 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, ⌊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 **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
    if (x < mid)
        return BinarySearch(A[a..⌊(a + b)/2⌋ - 1], x)
    else
        return BinarySearch(A[⌊(a + b)/2⌋ + 1..b], x)
```

Analysis: $T(n) = T(\lfloor n/2 \rfloor) + O(1)$. $T(n) = O(\log n)$.
Observation: After k steps, size of array left is $n/2^k$

Another common use of binary search

1. Optimization version: find solution of best (say minimum) value
2. 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):

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

Example

1. Problem: shortest paths in a graph.
2. Decision version: given $\mathbf{G}$ with non-negative integer edge lengths, nodes $\mathbf{s, t}$ and bound $\mathbf{B}$, is there an $\mathbf{s}$-$\mathbf{t}$ path in $\mathbf{G}$ of length at most $\mathbf{B}$?
3. 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?

1. Let $\mathbf{U}$ be maximum edge length in $\mathbf{G}$.
2. Minimum edge length is $\mathbf{L}$.
3. $\mathbf{s}$-$\mathbf{t}$ shortest path length is at most $\mathbf{(n-1)U}$ and at least $\mathbf{L}$.
4. Apply binary search on the interval $\mathbf{[L, (n-1)U]}$ via the algorithm for the decision problem.
5. $\mathbf{O(\log((n-1)U - L))}$ 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
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. **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*!

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. $\lim_{n\to\infty} \mathbf{F(n+1)/F(n)} = \phi$

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]
```

What is the running time of the algorithm? $O(n)$ 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. 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?

```
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 $\mathbf{i = 0, \ldots, n}$.

```
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]
```

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
        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.

Back to Fibonacci Numbers

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

1. input is n and hence input size is $\Theta(\log n)$
2. output is $F(n)$ and output size is $\Theta(n)$. Why?
3. Hence output size is exponential in input size so no polynomial time algorithm possible!
4. 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?
5. 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.

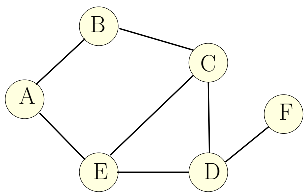
Part III

Brute Force Search, Recursion and Backtracking

Maximum Independent Set in a Graph

Definition

Given undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ 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 $(\mathbf{u}, \mathbf{v}) \not\in \mathbf{E}$.

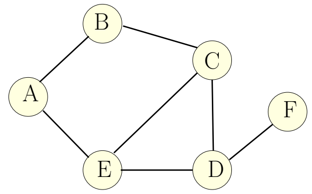


Some independent sets in graph above:

Maximum Independent Set Problem

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

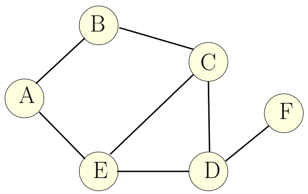
Goal Find maximum sized independent set in $\mathbf{G}$



Maximum Weight Independent Set Problem

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

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_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 = RecursiveMIS(G − v_n)
    b = w(v_n) + RecursiveMIS(G − v_n − N(v_n))
    Output max(a, b)
```


Recursive Algorithms

..for Maximum Independent Set

Running time:

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

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

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.

Example