Chapter 7

Binary Search, Introduction to Dynamic Programming

CS 473: Fundamental Algorithms, Spring 2011
February 8, 2011

7.1 Exponentiation, Binary Search

7.2 Exponentiation

7.2.0.1 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.

7.2.0.2 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

$$T(n) \leq T(\lfloor n/2 \rfloor) + 2$$

$T(n) = \Theta(\log n)$.

7.2.0.3 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.

7.2.0.4 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: $xy \mod p = ((x \mod p)(y \mod p)) \mod p$

7.2.0.5 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?).

7.3 Binary Search

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

7.3.0.7 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

7.3.0.8 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?

7.3.0.9 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.

7.4 Introduction to Dynamic Programming

7.4.0.10 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*

7.4.0.11 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.

7.5 Fibonacci Numbers

7.5.0.12 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*!

- $F(n) = (\phi^n - (1-\phi)^n)/\sqrt{5}$ where ϕ 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 $F(n)$.

7.5.0.13 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?

7.5.0.14 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.

7.5.0.15 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.

7.5.0.16 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)

7.5.0.17 Automatic explicit memoization

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

```
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

7.5.0.18 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
```

7.5.0.19 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.

7.5.0.20 Back to Fibonacci Numbers

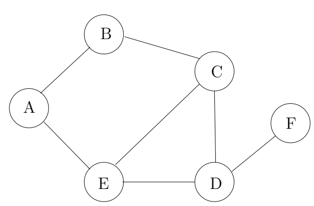
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- 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.

7.6 Brute Force Search, Recursion and Backtracking

7.6.0.21 Maximum Independent Set in a Graph

Definition 7.6.1 *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 $(u, v) \notin E$.*

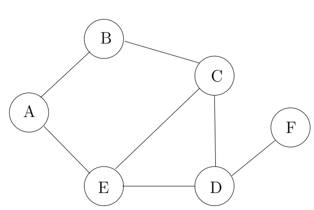


Some independent sets in graph above:

7.6.0.22 Maximum Independent Set Problem

Input Graph $G = (V, E)$

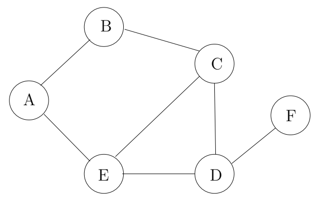
Goal Find maximum sized independent set in G



7.6.0.23 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



7.6.0.24 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.

7.6.0.25 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 $O(m2^n)$

7.6.0.26 A Recursive Algorithm

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

Observation 7.6.2 *v_n: Vertex in the grpah.*
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)
```

7.6.1 Recursive Algorithms

7.6.1.1 ..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 $T(n) = O(2^n)$.

7.6.1.2 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