ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ

ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

M.Sc. Program in Computer Science
Department of Informatics

Design and Analysis of Algorithms

Introduction

Vangelis Markakis

markakis@gmail.com

Content – Topics to be covered

- Introduction
  - Some basic concepts
  - Asymptotic notation: O, Ω, Θ
  - Distinction between polynomial, pseudopolynomial and exponential time algorithms
- Basic Algorithmic techniques
  - Divide and Conquer
  - Greedy algorithms
  - Dynamic Programming
  - And possibly some variations

Content – Topics to be covered

- Graph algorithms
  - Basics: DFS, BFS
  - Shortest paths, other path problems
  - Minimum Spanning trees, Matchings, Flow problems, TSP
- Complexity classes
  - The classes P and NP
  - NP-completeness reductions
- Linear and Integer Programming
  - Simplex and other algorithms for Linear Programming
  - Integer Programming in Combinatorial Optimization

Content – Topics to be covered

- Coping with NP-completeness – Approximation algorithms:
  - Definitions
  - Greedy and other combinatorial algorithms
  - LP-based algorithms
- Other types of algorithms (if time permits)
  - Randomized algorithms
  - Online algorithms

Bibliography

- [DPV] S. Dasgupta, C. H. Papadimitriou, U. V. Vazirani : "Algorithms"
- [KT] J. Kleinberg, E. Tardos: "Algorithm Design"
- [CLRS] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: "Introduction to Algorithms"
- …

- **and many resources on the WWW**

Communication

- Office hours (Derigny 12, 6th floor):
  - Tuesdays: 15:00 – 17:00
  - Thursdays: 14:00 – 15:00
- You can always email me regarding questions
  - If I do not reply within 3 days, send it again
- Eclass: Σχεδίαση και Ανάλυση Αλγορίθμων 2014-2015
  - Please check the announcements there at least once per week

Grading

- **4 Problem sets** **20%**
  - 1 set will be individual
  - 3 sets will be done in groups
  - 3 students per group
  - Each group will be examined 2 times on board
  - Each group will have to write and submit the problems on which they were examined
- **Midterm exam** **20%**
  - Probably on the 7th week, November 20
- **Final exam** **60%**

Tutorials

- Teaching Assistant: Giorgos Zois
- Office hours for the TA to be announced soon
- There will be 4 tutorial lectures
  - First one on Week 3
- Plus 3 slots for the homeworks' oral exams

Polynomial, Pseudo-Polynomial and Exponential Algorithms

What are we interested in?

Problems to be solved by a machine: precisely defined; no ambiguities

PROBLEM

INSTANCE (or INPUT): the (values of the) parameters of the problem
and possible constraints on them

QUESTION: what we are searching for

Algorithms

- Step-by-step procedures to solve problems using a machine
- Precise, unambiguous, mechanical procedures
- Independent of languages and systems
- The core of CS

Examples of Problems

EXP(onentiation)

I: positive integers a,n

Q: calculate a^n

FIBONACCI NUMBERS

I: a positive integer n

Q: calculate n-th Fibonacci number F_n

SUBSET SUM

I: a set $S=\{a_1, a_2, \ldots, a_n\}$ of n positive integers and an integer B

Q: is there a subset $A \subseteq S$ s. t. $\sum_{i \in A} a_i = B$?

SAT(isfiability)

I: a boolean formula ϕ

Q: Is ϕ satisfiable ?

(is there a value assignment to its variables making ϕ TRUE ?

= truth assignment)

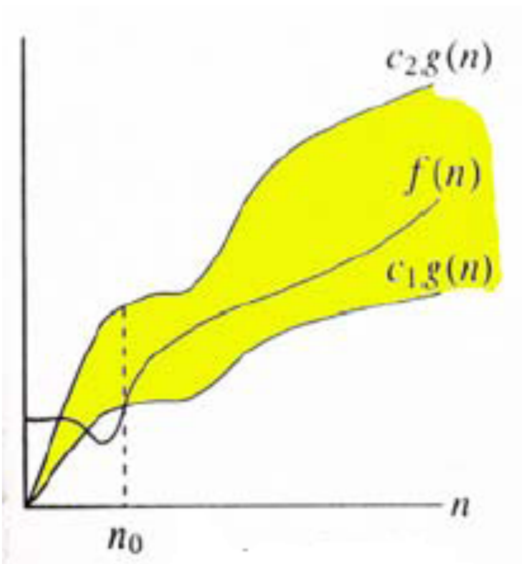
Algorithms

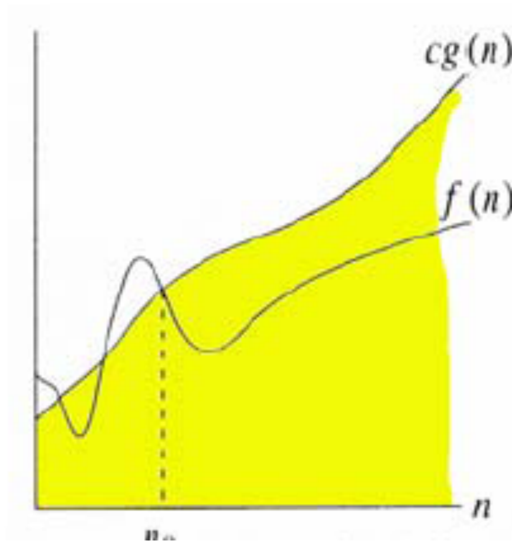
Three crucial questions about any algorithm for a problem:

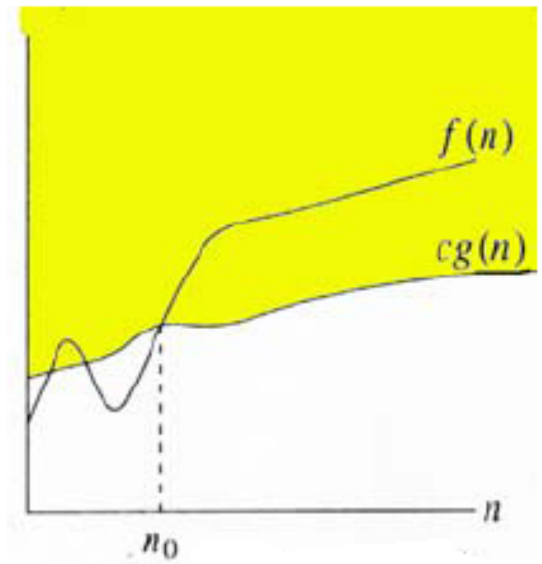
- **Is it correct ?**
  - Does it always terminate?
  - Does it give a correct answer for any instance of the problem ?
- **How much time/space it takes, as a function of its input?**
  - "time" = number of steps / "space" = number of bits in memory
  - "time" independent of language – implementation – machine
  - We mostly focus on time, expressed as a function T(n), where n is the size of the instance we try to solve
  - Interested in asymptotic behavior of T(n)
  - Notation: O, Ω, Θ, o, ω, $\approx$
- **Can we do better ?**

Asymptotic Notation

In pictures:







$f(n) = \Theta(g(n))$ $\quad$ $f(n) = O(g(n))$ $\quad$ $f(n) = \Omega(g(n))$

Asymptotic Notation

More formally:

- A function f(n) is O(g(n)) if there exist positive constants c_0 and n_0 such that $f(n) \leq c_0 g(n)$ for every $n \geq n_0$
  - The constant c_0 might be large (but still constant, **independent of** n)
  - Examples:
    - 2n + 10 is O(n). It suffices to set $c_0 = 3$ and $n_0 = 10$
    - 4nlogn + 150n + 3000sqrt(logn) = O(nlogn). $c_0 = 3154$, $n_0 = 1$

- A function f(n) is Ω(g(n)) if there exist positive constants c_0 and n_0 such that $f(n) \geq c_0 g(n)$ for every $n \geq n_0$
- A function f(n) is Θ(g(n)) if f(n) is O(g(n)) and f(n) is Ω(g(n))

Asymptotic Notation

- A polynomial of degree d is $O(n^d)$
  - Lower order terms do not matter
  - Constant terms also do not matter

- We tend to use the strictest possible bound
  - The function f(n) = 2n is O(n). It is also correct to say it is $O(n^2)$ but O(n) is a better upper bound

- Multiplying a term by a constant does not matter
  - 3n+5 = O(n). It is also O(3n) and O(n/2) but asymptotically there is no difference between O(n), O(3n), O(n/2)

Asymptotic Notation

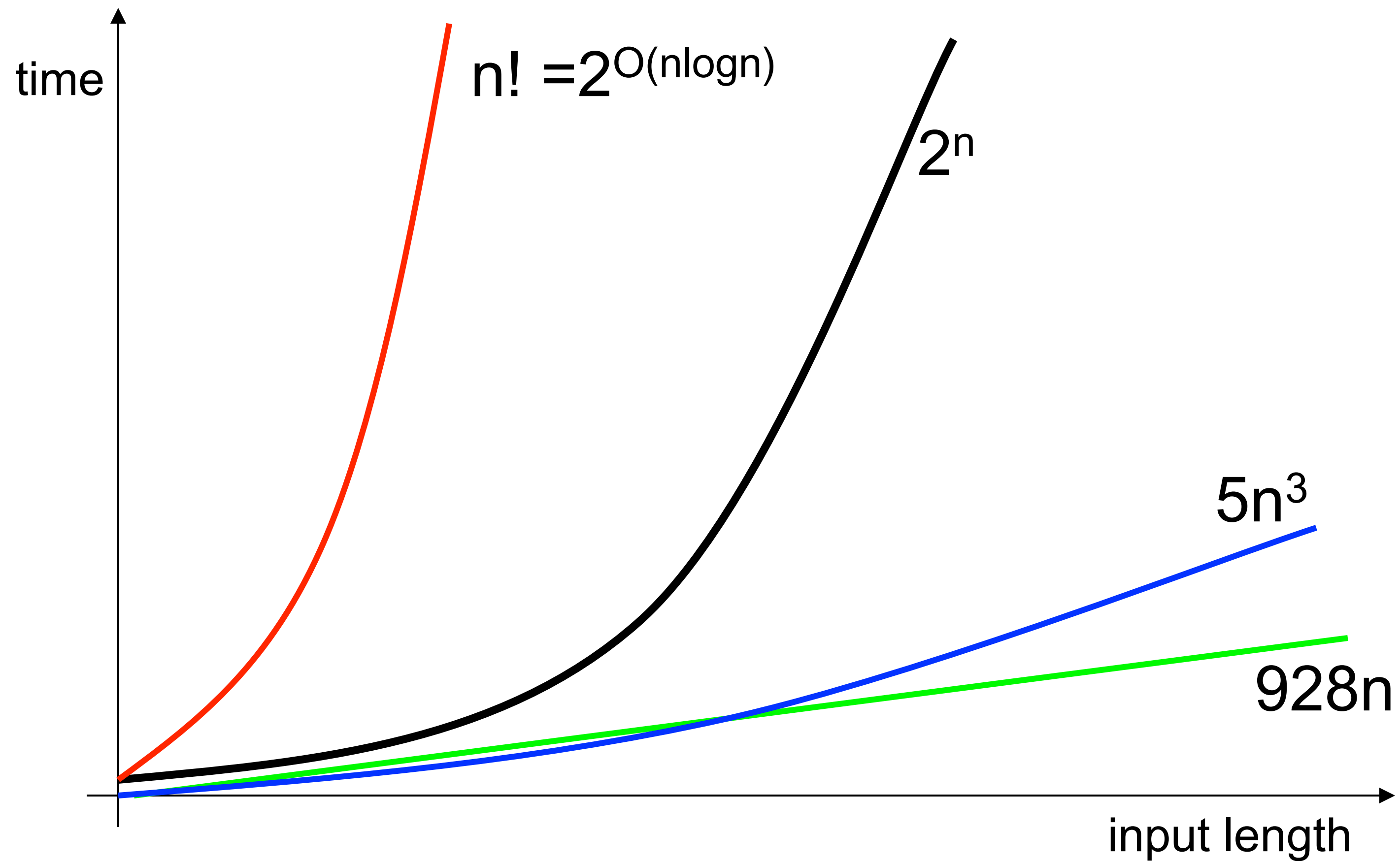
- All that matters is the largest term (the term that causes the greatest growth rate in the function)
  - If a function is $n^2+O(n)+O(n\log n)$, then it is $O(n^2)$

- Calculations with asymptotic terms: just as with regular numbers
  - $(n+O(1))(n+O(\log n) + O(1)) = n^2 + O(n\log n) + O(n)+ O(n) + O(\log n) + O(1) = n^2 + O(n\log n) + O(n) + O(\log n) + O(1) = O(n^2)$

More Examples

- $n^4 + 10n^3 + 80n^2 + 24 = O(n^4)$
- $2^n = O(2^{n+1})$ and $2^{n+1} = O(2^n)$
- $n^d = O(2^n)$ for every constant d (polynomial running times are always better than exponential ones)
- $2^n \neq O(n^d)$
- $(\log n)^m = O(n^d)$ for constants m, d
- $O(f(n))O(g(n)) = O(f(n)g(n))$

Growth of various functions



Time Complexity

There are many instances of the same size
How does the algorithm work over all these instances?

Best-case complexity

- The minimum number of steps taken on any instance of size n
- Not useful, too optimistic

Worst-case complexity

- The maximum number of steps taken on any instance of size n
- An upper bound on the complexity of the problem
- The most usual analysis

Average case complexity

- The average number of steps taken on any instance of size n
- Depends on the distribution of instances (use of probabilities)

Problem complexity and lower bounds

Complexity of a problem Π: $T_\Pi(n)$

The (worst case) complexity of the best (known) algorithm A

$$T_\Pi(n) = \min_A \{T_A(n)\}$$

Obtaining a lower bound on a problem's complexity $L_\Pi(n)$:

- By proving that there is no algorithm with $T_A(n) < L_\Pi(n)$
- Rare results, usually by restricting the model of computation (e.g., for sorting if we allow only number comparisons)

Optimal algorithm

- An algorithm A, for which $T_A(n) = L_\Pi(n)$
- For many problems we still do not know if we have found an optimal algorithm

Size of Instance and complexity

Consider the description of an instance (i.e., of all the parameters and constraints)

|I| = length of encoded instance/input

|I| = # of digits of the encoded input

Integer n:	Decimal	Binary	Unary
# bits :	$\lfloor \log_{10} n \rfloor + 1$	$\lfloor \log_2 n \rfloor + 1$	n

Size of Instance and complexity

|I| = length of encoded instance/input

|I| = # of digits of the encoded input (typically the binary encoding)

N(I) = the largest number in the input

Polynomial algorithms: O(poly(|I|)

Pseudo-Polynomial algorithms: O(poly(N(I)) that is O(exp(|I|)

We can say that they are O(poly(|I|)) <u>if we consider I encoded in unary !</u>

ONLY FOR PROBLEMS WITH NUMBERS !

Exponentiation

EXP(onentiation)

I: positive integers a,n
Q: calculate a^n

- Main operation in many cryptographic protocols (e.g., RSA)
- Very important to be able to compute this fast

```
Algorithm exp1(a,n);
//a, n positive integers
p:= 1;
for i:=1 to n do p:=p*a;
return p;
```

Correctness: obvious

Complexity: O(n)

If $a \le n$, then:

$|I| = \Theta(\log n) \Rightarrow n = \Theta(2^{|I|})$, $O(n)$ is $O(2^{|I|}) = O(\exp(I))$ NOT POLYNOMIAL !

$N(I) = n$, $O(n)$ is $O(poly(N(I)))$ PSEUDO-POLYNOMIAL !

Is there a polynomial algorithm for EXP ?

Exponentiation

$$a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{if } n \text{ is even} \\ a\left(a^{\left\lfloor \frac{n}{2} \right\rfloor}\right)^2 & \text{if } n \text{ is odd} \end{cases}$$

$$a^0 = 1$$

```
Algorithm exp2(a,n);
// a, n positive integers
if n = 0 then return 1;
z:= exp2(a, ⌊n/2⌋);
if n is even then return z^2
             else return az^2
```

Correctness: obvious

Complexity: O(logn), polynomial in |I| (why?)

T(n) = T(n/2) + O(1)

Fibonacci numbers

FIBONACCI

I: Recursive relation: $F_0 = 0$; $F_1 = 1$; $F_n = F_{n-1} + F_{n-2}$, $n \geq 2$

Q: Calculate F_n

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

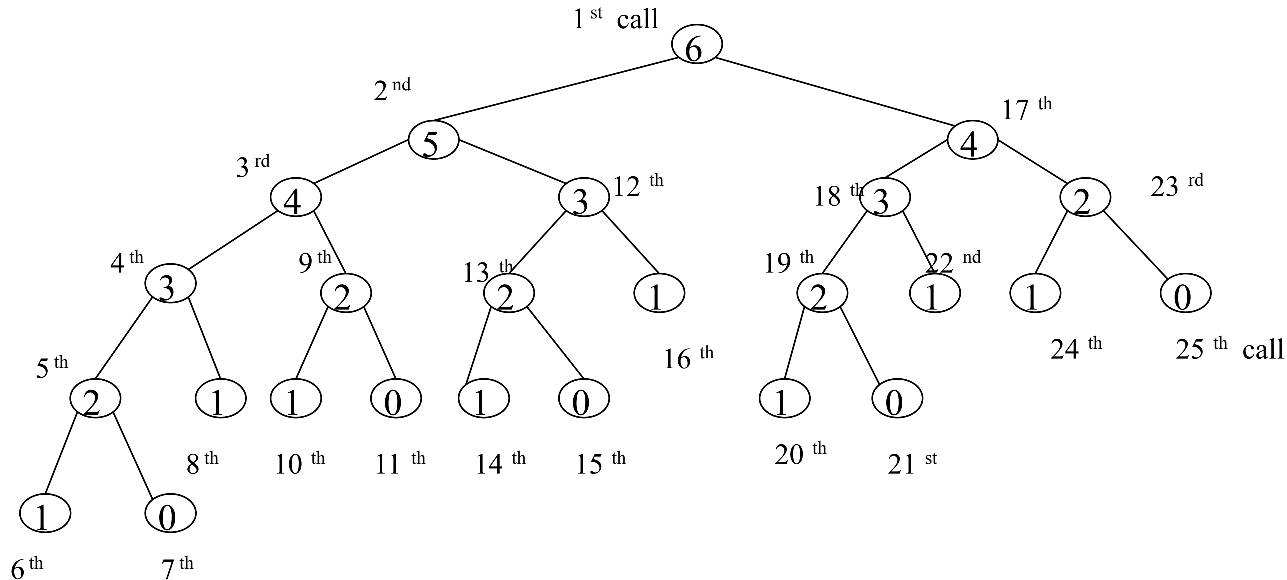
```
Algorithm fib1(n) // Direct implementation of recursion
   if n<2 then return n
          else return fib1(n-1)+ fib1(n-2)
```

Complexity of fib1(n): $T(0) = T(1) = 1$,

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

Fibonacci numbers

Call structure of recursion for n=6



Very inefficient: T(n) is $\Omega(2^{n/2})$

Full Binary Tree at least to depth n/2 → $2^{n/2}$ nodes

Source of inefficiency? It calulates the same values several times

Fibonacci numbers

```
Algorithm fib2(n) //remember already computed values
  f[0]:=0; f[1]:=1;
   for i:=2 to n do f[i]:= f[i-1] + f[i-2]
```

Complexity: O(n) NOT polynomial in |I| = O(logn)

Is there an O(logn) algorithm for F_n ?

Fibonacci numbers

Towards an O(logn) algorithm for F_n

Claim: It holds that (prove it)

$$F_n = \frac{1}{\sqrt{5}}\phi^n - \frac{1}{\sqrt{5}}\hat{\phi}^n, \text{ where } \phi = \frac{1+\sqrt{5}}{2} \text{ and } \hat{\phi} = \frac{1-\sqrt{5}}{2}$$

Exponentiation:

Hence, it suffices to compute φ^n and $\hat{\varphi}^n$

In fact, F_n is the closest integer to $\varphi^n / \sqrt{5}$, thus: $F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$ (why?)

- **Complexity:** O(logn) (why?) **but with use of irrational numbers…**
- Machines use finite arithmetic, irrational numbers may cause precision issues
- Is there an O(log n) algorithm for F_n without using irrational numbers ?

Fibonacci numbers

Claim: (prove it)

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Algorithm fib3(n): It suffices to compute

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$$

Complexity: O(logn) (why?)

Subset Sum

SUBSET SUM

I: a set $S = \{a_1, a_2, ..., a_n\}$ of n positive integers and an integer B

Q: is there a subset $A \subseteq S$ such that $\sum_{i \in A} a_i = B$?

BRUTE FORCE

- there are 2^n possible combinations of n items (= possible number of subsets one can construct from S)
- Go through all combinations and stop in the first one such that

$$\sum_{i \in A} a_i = B$$

- Report NO otherwise
- Running time: $O(n2^n)$

- Can we do better?

Subset Sum

RECALL COMPUTED VALUES

- Let $S_i=\{a_1, a_2,\ldots,a_i\}$
- IDEA: Compute the sums of all subsets of S_i using the sums of all subsets of S_{i-1}! (exclude sums > B)

- Let L be a list of integers
- **Notation:** L+b = a new list with all elements of L increased by b

  e.g., if L = [1, 2, 3, 5], L+2 = [3, 4, 5, 7]

We will use an auxiliary algorithm for merging sums of subsets:

- MERGE (L,L')
  - Input: 2 sorted lists of integers, L and L'
  - Returns a sorted list that is the merge of the sorted lists L and L' with no duplicate values
  - Complexity $O(|L|+|L'|)$ (why ?)

Subset Sum

L_i : list of the sums of all subsets of S_i (keep only sums ≤ B)

```
Algorithm SubsetSum (S,B);
L0=[0];
for i=1 to n do
   Li=MERGE(Li-1, Li-1+ai);
   Remove from Li every element >B;
Check if the largest element in L equals B;
```

Example

S={1,4,5}, n=3, B=8

L_0=[0] L_0+a_1 =[1]

L_1=[0,1] L_1+a_2 =[4,5]

L_2=[0,1,4,5] L_2+a_3 =[5,6,9,10]

L_3=[0,1,4,5,6] Answer: NO

Complexity ?

Subset Sum

Complexity: O(nB)

At every step, the list we keep has at most B elements

- How big is O(nB) with respect to |I|?
- The input I consists of n+1 numbers: a_1, a_2, ..., a_n, B
- Let N(I) be the maximum number $\Rightarrow$ $|I| \leq (n+1)\log(N(I))$
- In worst case, N(I) = B $\Rightarrow$ $B \geq 2^{|I|/(n+1)}$
- O(nB) is O(exp(I)) NOT POLYNOMIAL !
- BUT: O(nB) is O(poly(N(I)) PSEUDO-POLYNOMIAL !

Can we do better ? (probably) NO !

Is there a polynomial algorithm for SUBSET SUM ? (probably) NO !

Boolean Formulas and SAT

Boolean variables: T(RUE) / F(ALSE) or 1 / 0

Boolean operators: AND ($x \wedge y$), OR ($x \vee y$), NOT ($\neg x$ / $\bar{x}$)

Literal: Boolean variable (x) or its negation ($\neg x$ or $\bar{x}$)

Boolean formula: $\phi(x,y) = (\neg x \vee y) \wedge (x \vee \neg y)$

SAT
I: a boolean formula ϕ
Q: Is ϕ *satisfiable* ?
(is there a value assignment to its variables making ϕ TRUE ?
= Truth Assignment)

Example: $\phi(x,y) = (\neg x \vee y) \wedge (x \vee \neg y)$ is satisfiable
by the assignments x=y=T
and x=y=F

CNF-SAT

- **Clause =** A set of OR-ed literals, e.g. (x ∨ ¬y ∨ z)
- A formula is in Conjunctive Normal Form (CNF) if:
  - it is the AND of a set of clauses

E.g. $\phi = (w \vee x \vee y \vee z),\ (w \vee \overline{x}),\ (x \vee \overline{y}),\ (y \vee \overline{z}),\ (z \vee \overline{w}),\ (\overline{w} \vee \overline{z})$

Any formula ϕ can be written in CNF

(CNF)-SAT

I: a boolean formula ϕ in CNF

Q: Is ϕ *satisfiable* ?

SAT

Brute-force approach

- there are 2^n possible assignments for n variables
- Go through all possible assignments;
  stop in the first truth assignment or report NO
- Running time: $O(2^n)$

Backtracking:

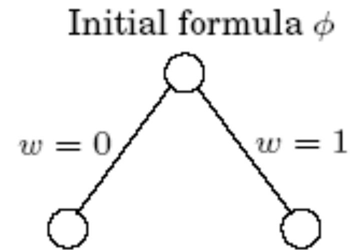
- Intelligent exhaustive search
- Consider partial assignmnets
- Prune the search space
- Example:

$$\phi = (w \vee x \vee y \vee z),\ (w \vee \overline{x}),\ (x \vee \overline{y}),\ (y \vee \overline{z}),\ (z \vee \overline{w}),\ (\overline{w} \vee \overline{z})$$

SAT

Start with the initial formula

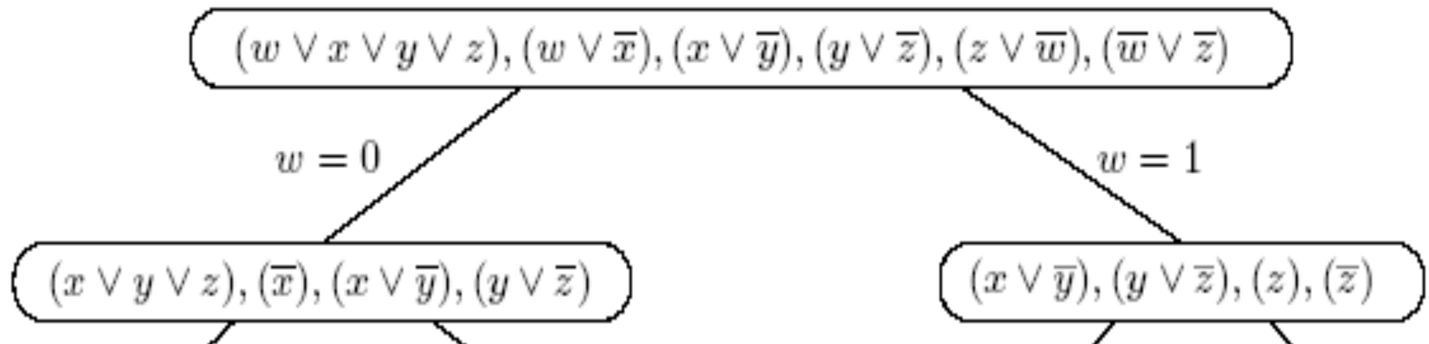
Branch on a variable, e.g. w



Plug into ϕ the values of w

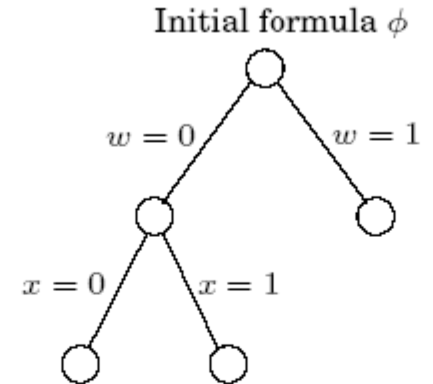
No clause is immediately violated

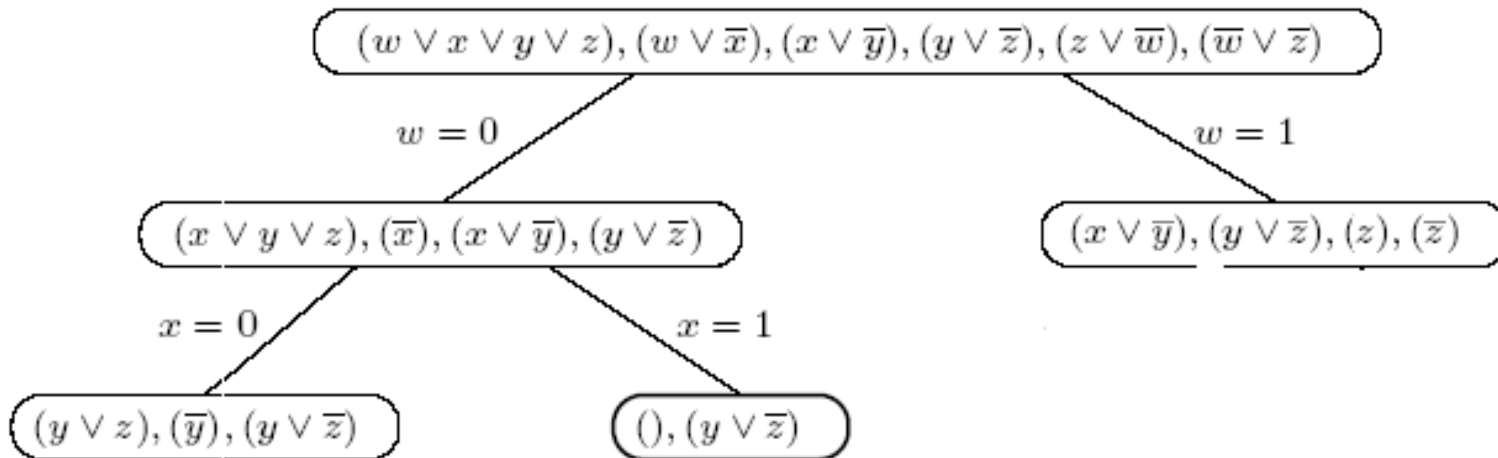
Keep active both branches



SAT

Expand an active node on a new variable, e.g. x

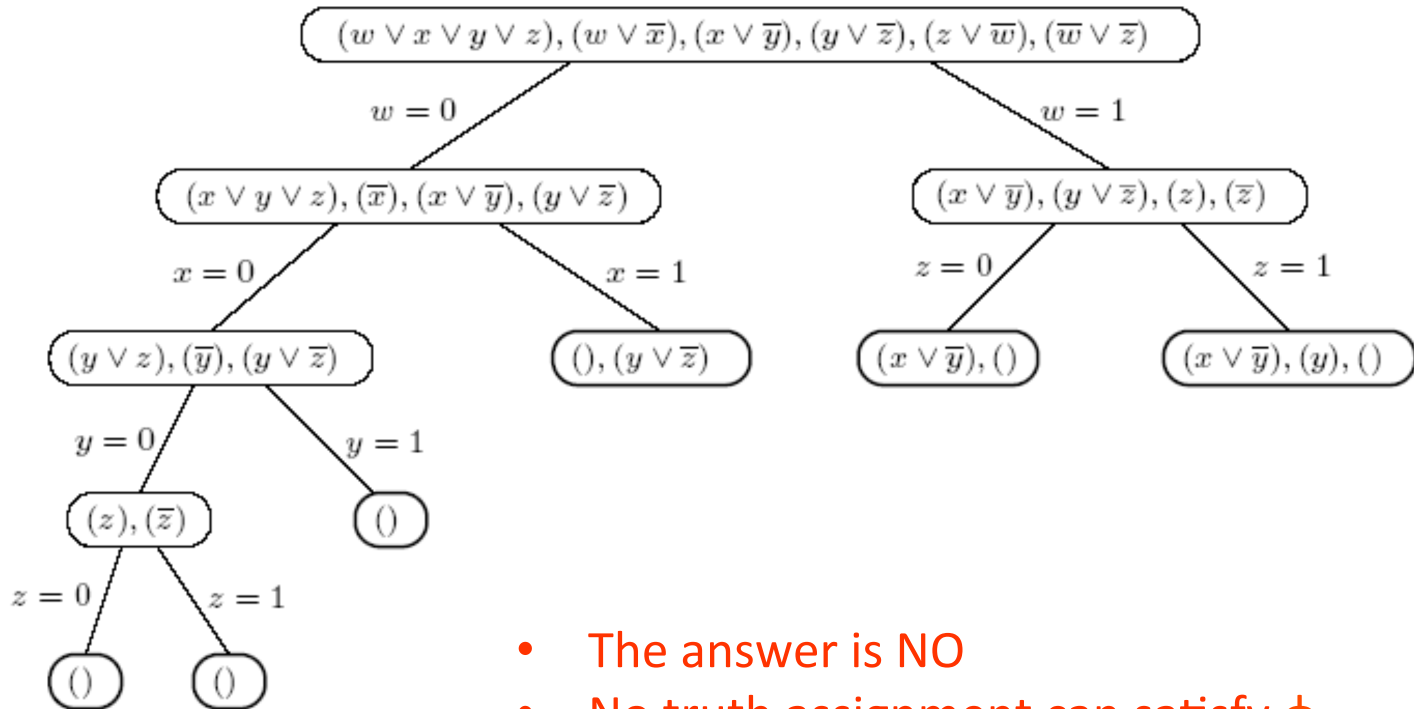




( ): FALSE clause; Do not expand, the partial assignment cannot make ϕ satisfiable

SAT

Finally:



- The answer is NO
- No truth assignment can satisfy ϕ
- Did not have to search all assignments

SAT

Worst case complexity: We may have to explore all possible branchings: $O(2^n)$!

Can we do better ?

Yes, but still exponentially!

Among the most well-studied problems in the literature:

$1{,}839^n$, $1{,}769^n$, $1{,}619^n$, …

Is there a polynomial algorithm for SAT ? (probably) NO !

Is there a pseudo-polynomial algorithm for SAT ? NO !

(SAT is not a problem with numbers !)

Summary

Problem	Algorithms (complexity)		
	exp(|I|)	poly(N(I)) (pseudo-poly)	poly(|I|)
Exponentiation	--	O(n)	O(log n)
Fibonacci numbers	$\Omega(2^{n/2})$	O(n)	O(log n)
SUBSET SUM	$O(2^n)$	O(nB)	NO *
SAT	$O(2^n)$	--	NO *

* Unless P=NP