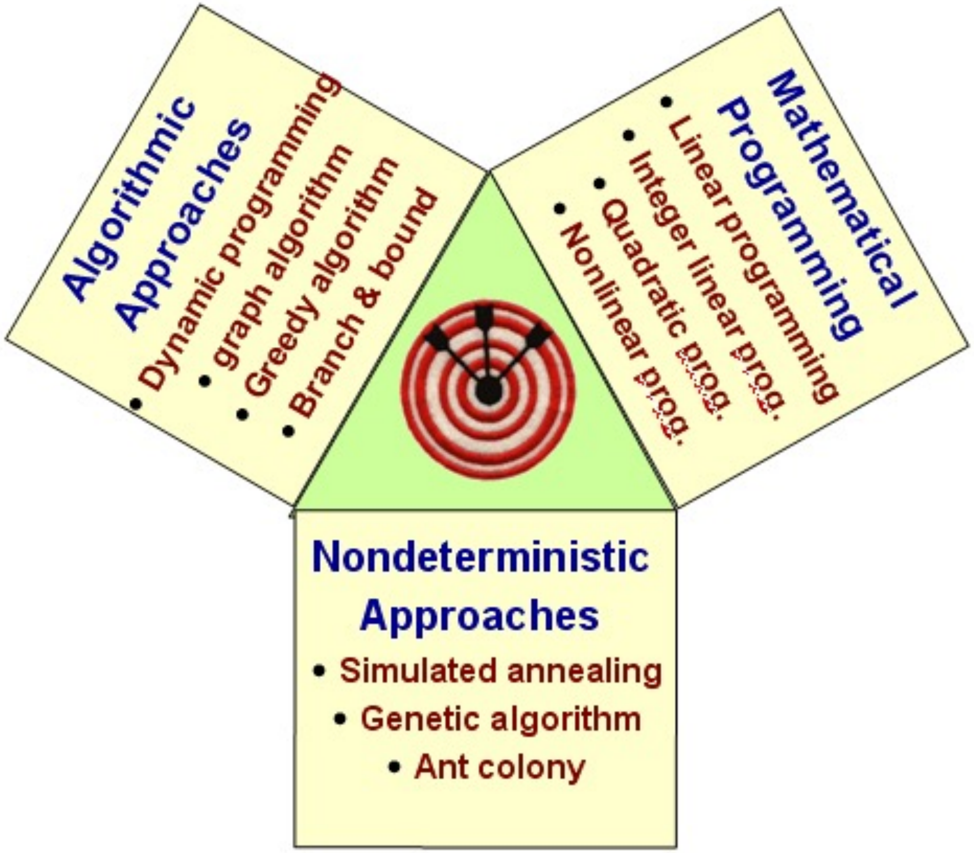
Unit 2: EDA Paradigms & Complexity

- Course contents:
  - Computational Complexity
  - EDA paradigms: Algorithms, Frameworks, Methodology
- Readings
  - W&C&C: Chapter 4
  - S&Y: Appendix A



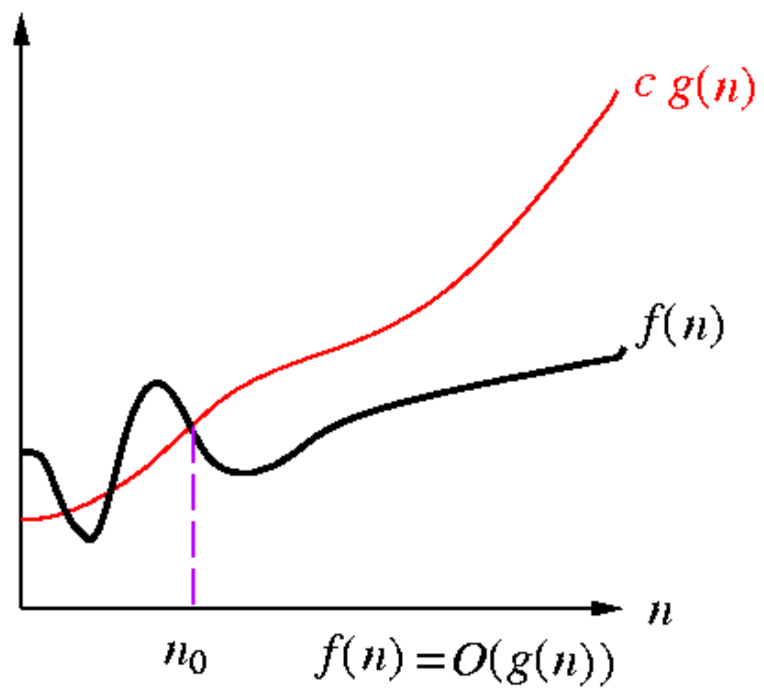
Why Does Complexity Matter?

- To characterize the efficiency/hardness of problem solving
- Have a better idea on how to come up with good algorithms
  - Algorithm: a well-defined procedure transforming some *input* to a desired *output* in **finite** computational resources in *time* and *space*.
- Runtime comparison: assume 1 BIPS (Billion Instructions Per Sec.), 1 instruction/operation.

Time	Big-Oh	$n = 10$	$n = 100$	$n = 10^4$	$n = 10^6$	$n = 10^8$
500	$O(1)$	$5*10^{-7}$ sec	$5*10^{-7}$ sec	$5*10^{-7}$ sec	$5*10^{-7}$ sec	$5*10^{-7}$ sec
$3n$	$O(n)$	$3*10^{-8}$ sec	$3*10^{-7}$ sec	$3*10^{-5}$ sec	0.003 sec	0.3 sec
$n \lg n$	$O(n \lg n)$	$3*10^{-8}$ sec	$6*10^{-7}$ sec	$1*10^{-4}$ sec	0.018 sec	2.5 sec
n^2	$O(n^2)$	$1*10^{-7}$ sec	$1*10^{-5}$ sec	0. 1 sec	16.7 min	116 days
n^3	$O(n^3)$	$1*10^{-6}$ sec	0.001 sec	16.7 min	31.7 yr	∞
2^n	$O(2^n)$	$1*10^{-6}$ sec	$4*10^{11}$ cent.	∞	∞	∞
$n!$	$O(n!)$	0.003 sec	∞	∞	∞	∞

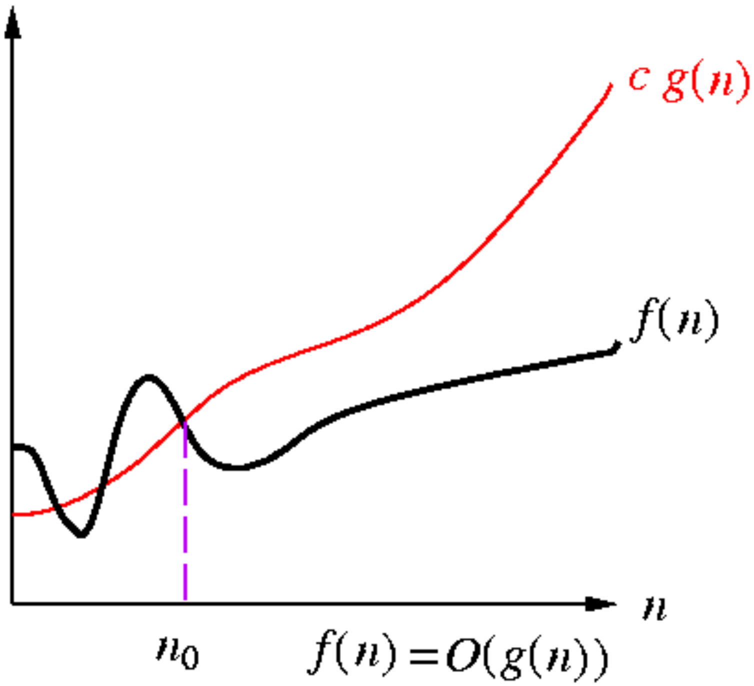
O: Upper Bounding Function

- **Def:** $f(n) = O(g(n))$ if $\exists\, c > 0$ and $n_0 > 0$ such that $0 \le \boldsymbol{f(n)} \le \boldsymbol{cg(n)}$ for all $n \ge n_0$.
  - Examples: $2n^2 + 3n = O(n^2)$, $2n^2 = O(n^3)$, $3n \lg n = O(n^2)$
- Intuition: $f(n)$ "$\le$" $g(n)$ when we ignore constant multiples and small values of n.



Big-O Notation

- "An algorithm has worst-case running time $O(f(n))$": there is a constant c s.t. for every n big enough, **every execution** on an input of size n takes **at most** $cf(n)$ time.
- Only the dominating term needs to be kept while constant coefficients are immaterial.
  - e.g.,
    $0.3n^2 = O(n^2)$,
    $3n^2 + 152n + 1777 = O(n^2)$,
    $n^2 \lg n + 3n^2 = O(n^2 \lg n)$
  - The following are correct but not used
    $3n^2 = O(n^2 \lg n)$
    $3n^2 = O(0.1n^2)$
    $3n^2 = O(n^2 + n)$

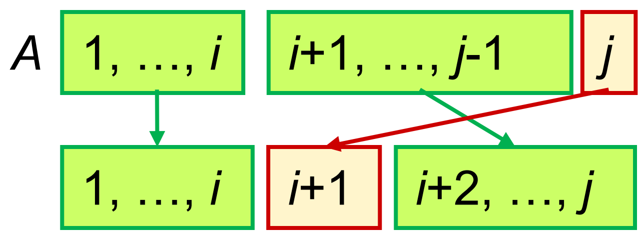


Computational Complexity

- Computational complexity: an abstract measure of the time and space necessary to execute an algorithm as function of its "input size".
- Input size examples:
  - sort n words of bounded length ⍰ n
  - the input is the integer n ⍰ $\lg n$
  - the input is the graph $G(V, E)$ ⍰ $|V|$ and $|E|$
- Time complexity is expressed in *elementary computational steps* (e.g., an addition, multiplication, pointer indirection).
- Space Complexity is expressed in *memory locations* (e.g. bits, bytes, words).

Example: Insertion Sort

- Idea:
  - Insert a number $A[j]$ into a sorted sequence with j-1 numbers
  - Result in a sorted sequence with j numbers

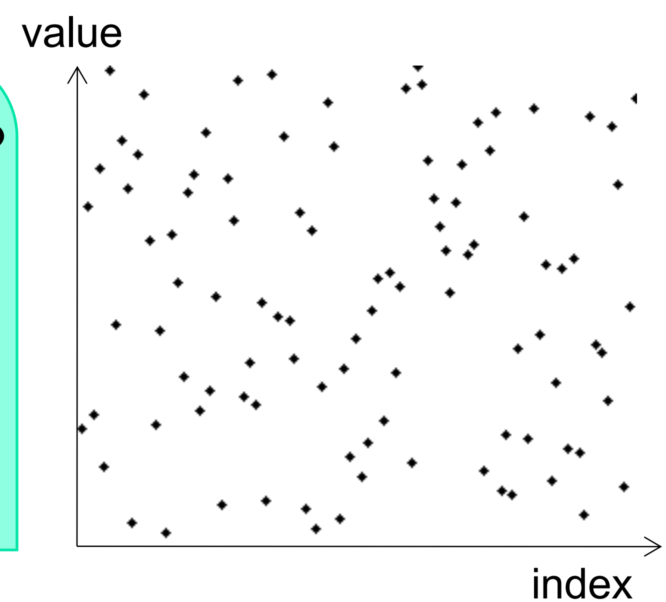


Time complexity?

```
InsertionSort(A)
for j = 2 to length[A] do
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1]
    i = j-1
    while i > 0 and A[i] > key do
        A[i+1] = A[i]
        i = i -1
    A[i+1] = key
```

$O(j^2)$

$O(j)$



http://en.wikipedia.org/wiki/Image:Insertion_sort_animation.gif

Mark P.-H. Lin [Courtesy of Y.-W. Chang, H.-M Chen]

Amortized Analysis

- **Why Amortized Analysis?**
  - Find a tight bound of a sequence of data structure operations.
- No probability involved, guarantees the **average performance of each operation in the worst case**
- Three popular methods
  - Aggregate method
  - Accounting method
  - Potential method

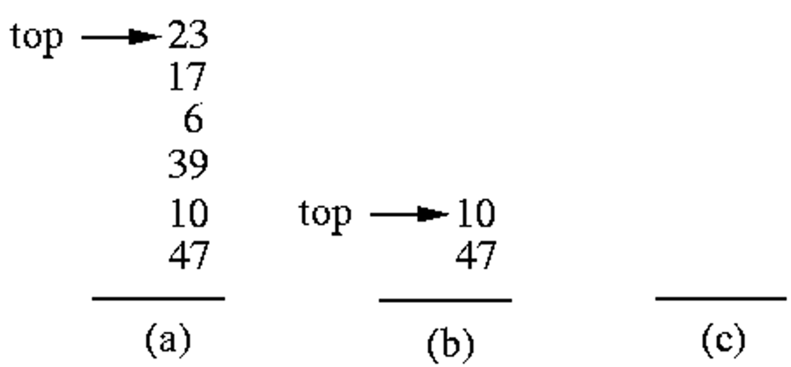
Methods for Amortized Analysis

- **Aggregate** method
  - n operations take $T(n)$ time.
  - Average cost of an operation is $T(n)/n$ time.
- **Accounting** method
  - Charge each type of operation an amortized cost.
  - Store the overcharge of early operations as "prepaid credit" in "bank."
  - Use the credit for later operations.
  - **Must guarantee nonnegative balance at all time**
- **Potential** method
  - View "prepaid credit" as "potential energy."

Aggregate Method: Stack and MULTIPOP

- n operations take $T(n)$ time $\Rightarrow$ average cost of an operation is $T(n)/n$ time.
- Consider a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack.
  - Worst-case analysis: a MULTIPOP operation takes $O(n)$.
  - Aggregate method: Any sequence of n PUSH, POP, MULTIPOP costs at most $O(n)$ time **(why?)** $\Rightarrow$ amortized cost of an operation: $O(n)/n=O(1)$.

```
Multipop(S, k)
while not Stack-Empty(S) and k > 0 do
    Pop(S)
    k ← k-1
```



Multipop(S, 4) Multipop(S, 7)

Accounting Method

- Assign differing charges to different operations.
  - Amortized cost = actual cost + credit.
  - Credit is assigned to specific objects and must be nonnegative all the times.
- Stack operations (s: stack size):

	Actual cost	Amortized cost
PUSH	1	2
POP	1	0
MULTIPOP	Min(k, s)	0

  - For any sequence of n operations, total actual cost $\leq$ total amortized cost = $O(n)$.

Potential Method

- Represent the prepaid work as "potential" that can be released to pay for future operations.
  - Potential is associated with the whole data structure, not with specific items in the data structure. (cf. accounting method)
- The potential method:
  - D_0: initial data structure
    D_i: data structure after applying the *i*th operation to D_{i-1}
    c_i: actual cost of the *i*th operation
  - Define the potential function $\Phi : D_i \rightarrow \Re$.
  - Amortized cost $\hat{c}_i$, $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

  - Pick $\Phi(D_n) \geq \Phi(D_0)$ to make $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$
  - Often define $\Phi(D_0) = 0$ and then show that $\Phi(D_i) \geq 0, \forall i$.

Potential Method: Stack Operations

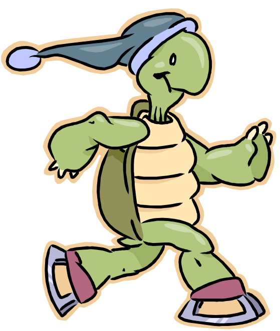
- Amortized cost of each operation = $O(1)$.
- $\Phi(D)$ = # of objects in the stack D; $\Phi(D_0)=0$, $\Phi(D_i) \geq 0$.
- PUSH: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s+1) - s = 2$.
- POP: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (-1) =$ **0**.
- MULTIPOP(S, k): $k' = \min(k, s)$ objects are popped off.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0.\end{aligned}$$

Asymptotic Functions

- Polynomial-time complexity: $O(n^k)$, where n is the **input size** and k is a constant.
- Example polynomial functions:
  - 999: constant
  - $\lg n$: logarithmic
  - $\sqrt{n}$: sublinear
  - n: linear
  - $n \lg n$: loglinear
  - n^2: quadratic
  - n^3: cubic
- Example non-polynomial functions
  - 2^n, 3^n: exponential
  - $n!$: factorial

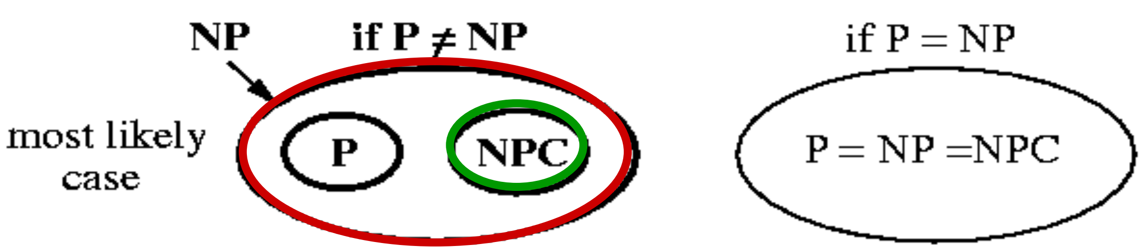
Faster



Slower

Complexity Classes

- **Class P:** class of problems that can be **solved** in polynomial time in the **size of input**.
  - Edmonds: Problems in P are considered **tractable**.
- **Class NP** (**N**ondeterministic **P**olynomial): class of problems whose solutions can be **verified** in polynomial time in the size of input.
  - $P \subset NP$ or $P = NP$?
- **Class NP-complete (NPC): Any** NPC problem can be solved in polynomial time ➔ **all** problems in NP can be solved in polynomial time (i.e., P = NP).



Coping with a "Tough" Problem: Trilogy I

"I can't find an efficient algorithm.
I guess I'm just too dumb."

Coping with a "Tough" Problem: Trilogy II

"I can't find an efficient algorithm,
because no such algorithm is possible!"

Coping with a “Tough” Problem: Trilogy III

“I can’t find an efficient algorithm,
but neither can all these famous people.”

Algorithmic Paradigms

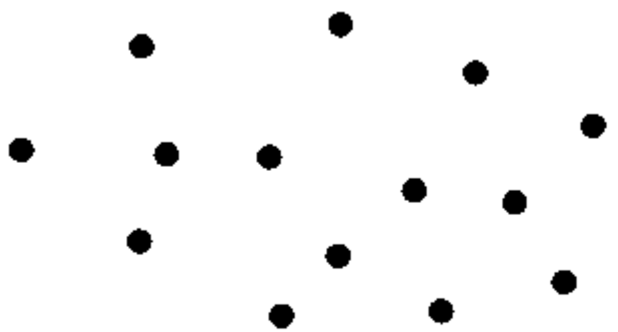
- **Exhaustive search**: Search the entire solution space
- **Branch and bound**: Search with pruning
- **Greedy**: Pick a locally optimal solution at each step
- **Dynamic programming**: if subproblems are not independent
- **Divide-and-conquer** (a.k.a. **hierarchical**): Divide a problem into subproblems (small and similar), solve subproblems, and then combine the solutions of subproblems
- **Mathematical programming**: Solve an objective function under constraints
- **Local search**: Move from solution to solution in the search space until a solution deemed optimal is found or a time bound is elapsed
- **Probabilistic**: Make some choices randomly (or pseudo-randomly
- **Reduction**: Transform into a known and optimally solved problem

Algorithm Types

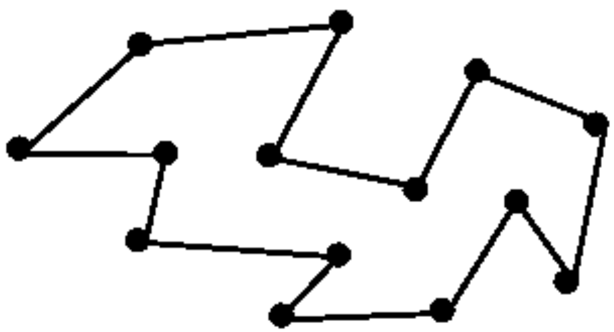
- Algorithms usually used for P problems
  - Exhaustive search
  - Branch and bound
  - Divide-and-conquer (a.k.a. hierarchical)
  - Dynamic programming
  - Mathematical programming
- Algorithms usually used for NP (but not P) problems
  (strategy: not seeking an "optimal solution", but a "good" one)
  - Approximation
  - Pseudo-polynomial time: polynomial form, but NOT to input size
  - Restriction: restrict the problem to a special case that is in P
  - Exhaustive search/branch and bound
  - Local search: simulated annealing, genetic algorithm, ant colony
  - Heuristics: greedy, … etc

The Traveling Salesman Problem (TSP)

- **Instance:** a set of n cities, a distance between each pair of cities, and a bound B.
- **Question:** is there a route that starts and ends at a given city, visits every city exactly once, and has total distance $\leq B$?



A TSP instance



A TSP solution

NP vs. P

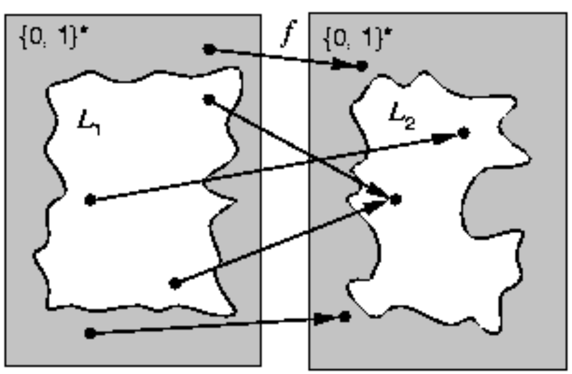
- TSP ∈ NP.
  - Need to **check** a solution (tour) in polynomial time.
    - Guess a tour.
    - Check if the tour visits every city exactly once, returns to the start, and total distance ≤ *B*.
- TSP ∈ P?
  - Need to solve (find a tour) in polynomial time.
  - Still unknown if TSP ∈ P.

Decision Problems and NP-Completeness

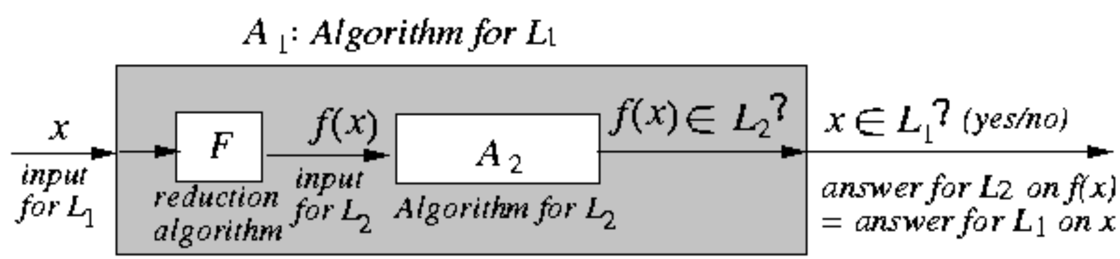
- **Decision problems:** those having yes/no answers.
  - TSP: Given a set of cities, distance between each pair of cities, and a bound *B*, **is there a route** that starts and ends at a given city, visits every city exactly once, and has total distance at most *B*?
- **Optimization problems:** those finding a legal configuration such that its cost is minimum (or maximum).
  - TSP: Given a set of cities and that distance between each pair of cities, **find the distance of a "minimum route"** that starts and ends at a given city and visits every city exactly once.
- Could apply binary search on decision problems to obtain solutions to optimization problems.
- NP-completeness is associated with decision problems.

Polynomial-time Reduction

- **Motivation:** Let *L*1 and *L*2 be two decision problems.
  Suppose algorithm *A*2 can solve *L*2. Can we use *A*2 to solve *L*1?
- **Polynomial-time reduction *f* from *L*1 to *L*2:** $L1 \le_P L2$
  - *f* reduces input for *L*1 into an input for *L*2 s.t. the reduced input is a "yes" input for *L*2 iff the original input is a "yes" input for *L*1.
    - $L1 \le_P L2$ if $\exists$ polynomial-time computable function f: $\{0, 1\}^* \rightarrow \{0, 1\}^*$ s.t. $x \in L1$ iff $f(x) \in L2$, $\forall\, x \in \{0, 1\}^*$.
    - ***L*2 is at least as hard as *L*1.**
- *f* is computable in polynomial time.

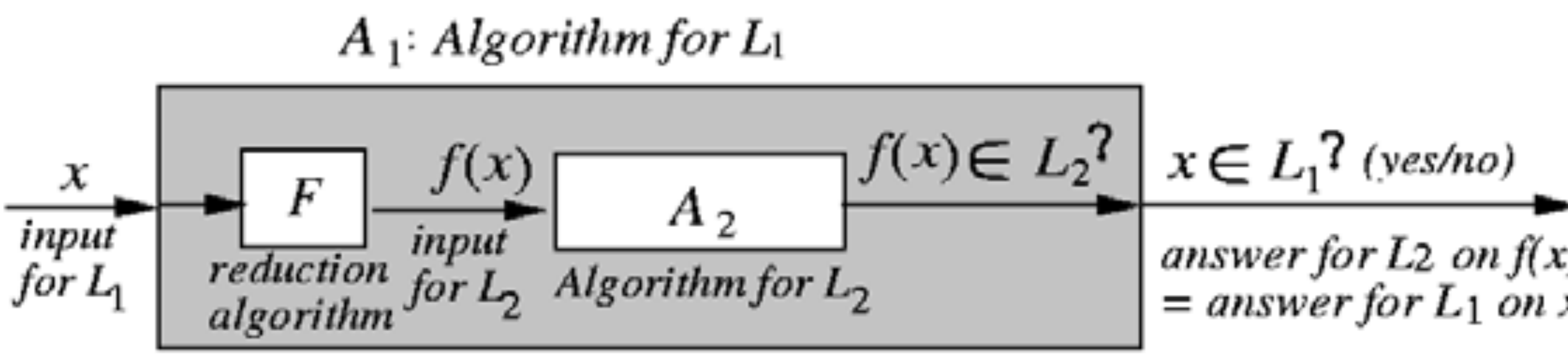


$L_1 \le_P L_2$



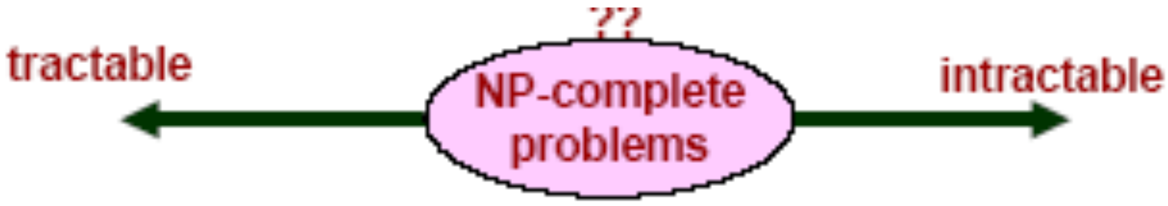
Significance of Reduction

- Significance of $L1 \le_P L2$:
  - $\exists$ polynomial-time algorithm for $L2 \Rightarrow \exists$ polynomial-time algorithm for $L1$ ($L2 \in \mathrm{P} \Rightarrow L1 \in \mathrm{P}$).
  - polynomial-time algorithm for $L1 \Rightarrow$ polynomial-time algorithm for $L2$ ($L1 \notin \mathrm{P} \Rightarrow L2 \notin \mathrm{P}$).
- $\le_P$ is transitive, i.e., $L1 \le_P L2$ and $L2 \le_P L3 \Rightarrow L1 \le_P L3$.



NP-Completeness

- **NP-completeness: worst-case** analyses for **decision** problems.
- A **decision** problem L is **NP-complete (NPC)** if
  1. $L \in$ NP, and
  2. $L' \leq_P L$ for every $L' \in$ NP.
- **NP-hard:** If L satisfies property 2, but not necessarily property 1, we say that L is **NP-hard**.
- Suppose $L \in$ NPC.
  - If $L \in P$, then there exists a polynomial-time algorithm for every $L' \in$ NP (i.e., P = NP).
  - If $L \notin P$, then there exists no polynomial-time algorithm for any $L' \in$ NPC (i.e., P ≠ NP).



Proving NP-Completeness

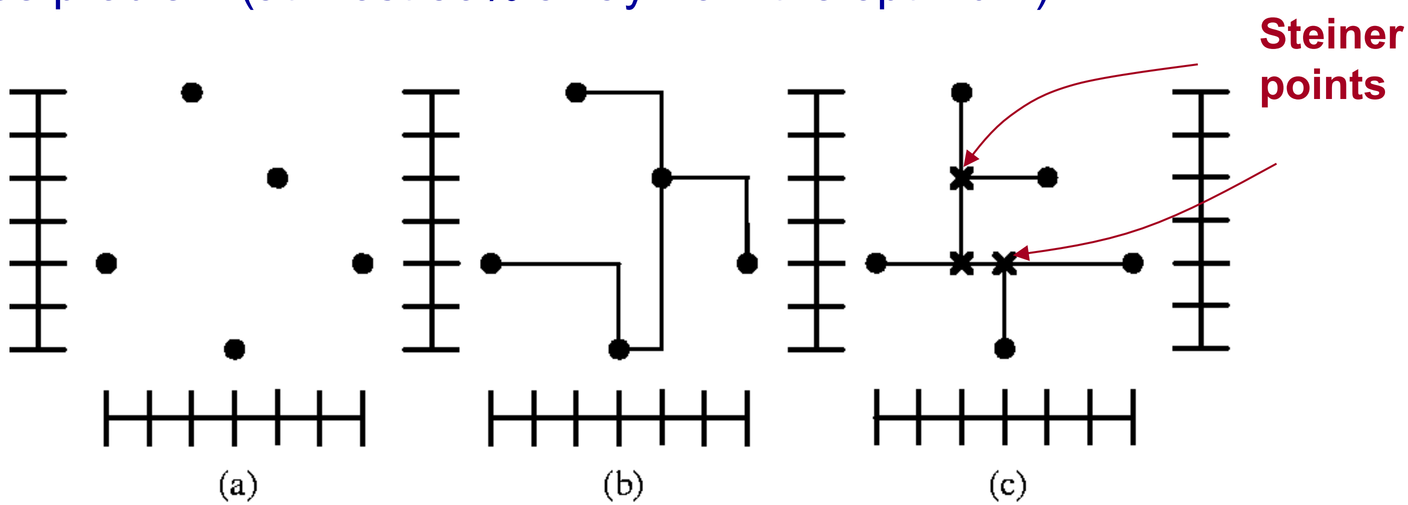
- **Five steps for proving that *L* is NP-complete:**
  1. Prove $L \in$ NP. (easy)
  2. Select a known NP-complete problem ***L'***.
  3. Construct a reduction *f* transforming **every** instance of ***L'*** to an instance of *L*.
  4. Prove that $x \in$ ***L'*** iff $f(x) \in L$ for all $x \in \{0, 1\}^*$.
  5. Prove that *f* is a polynomial-time transformation.

Coping with NP-hard Problems

- **Exhaustive search/Branch and bound**
  - Is feasible only when the problem size is small.
- **Approximation algorithms**
  - Guarantee to be a fixed percentage away from the optimum.
  - E.g., MST for the minimum Steiner tree problem.
- **Pseudo-polynomial time algorithms**
  - Has the form of a polynomial function for the complexity, but is not to the problem size.
  - E.g., $O(nW)$ for the 0-1 knapsack problem. (W: maximum weight)
- **Restriction**
  - Work on some subset of the original problem.
  - E.g., the maximum independent set problem in circle graphs.
- **Heuristics:** No guarantee of performance.

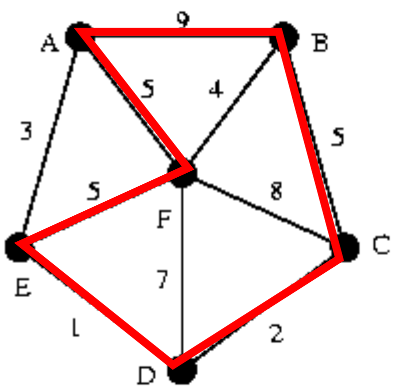
Spanning Tree vs. Steiner Tree

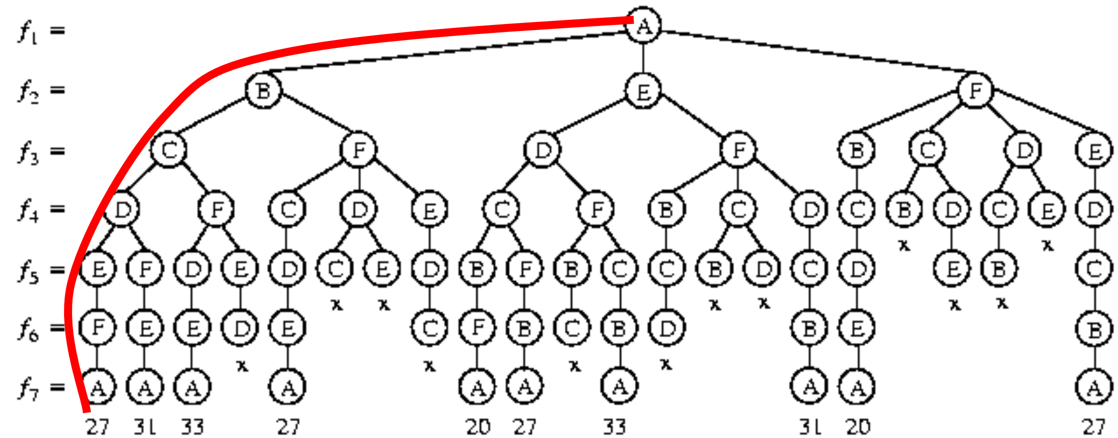
- **Manhattan distance:** If two points (nodes) are located at coordinates (x_1, y_1) and (x_2, y_2), the Manhattan distance between them is given by $d_{12} = |x_1 - x_2| + |y_1 - y_2|$ (a.k.a. λ-1 metric)
- **Rectilinear spanning tree:** a spanning tree that connects its nodes using Manhattan paths (Fig. (b) below).
- **Steiner tree:** a tree that connects its nodes, and additional points (**Steiner points**) are permitted to be used for the connections.
- The minimum rectilinear spanning tree problem is in P, while the minimum rectilinear Steiner tree (Fig. (c)) problem is NP-complete.
  - The spanning tree algorithm can be an *approximation* for the Steiner tree problem (at most 50% away from the optimum).



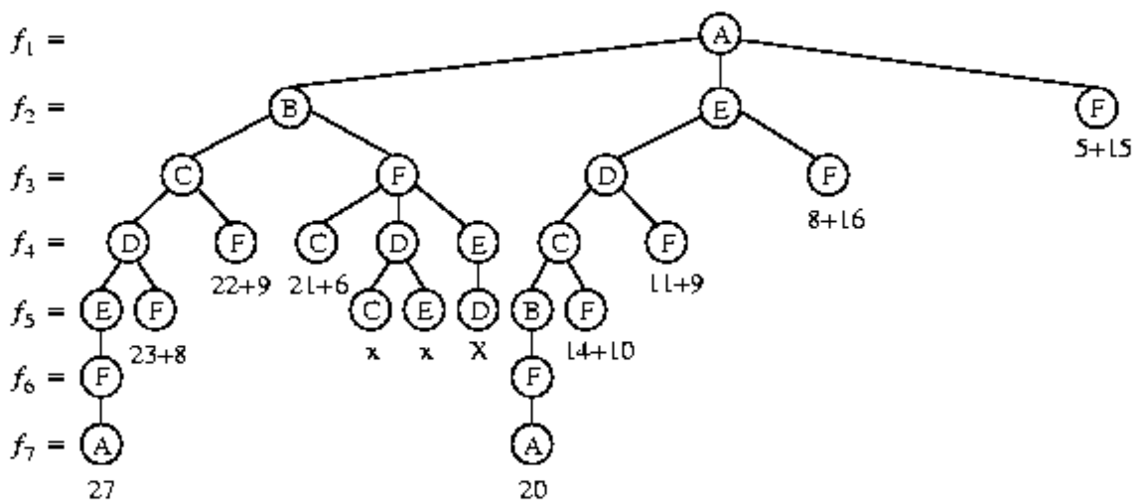
Exhaustive Search vs. Branch and Bound

- TSP example





Backtracking/exhaustive search



Branch and bound

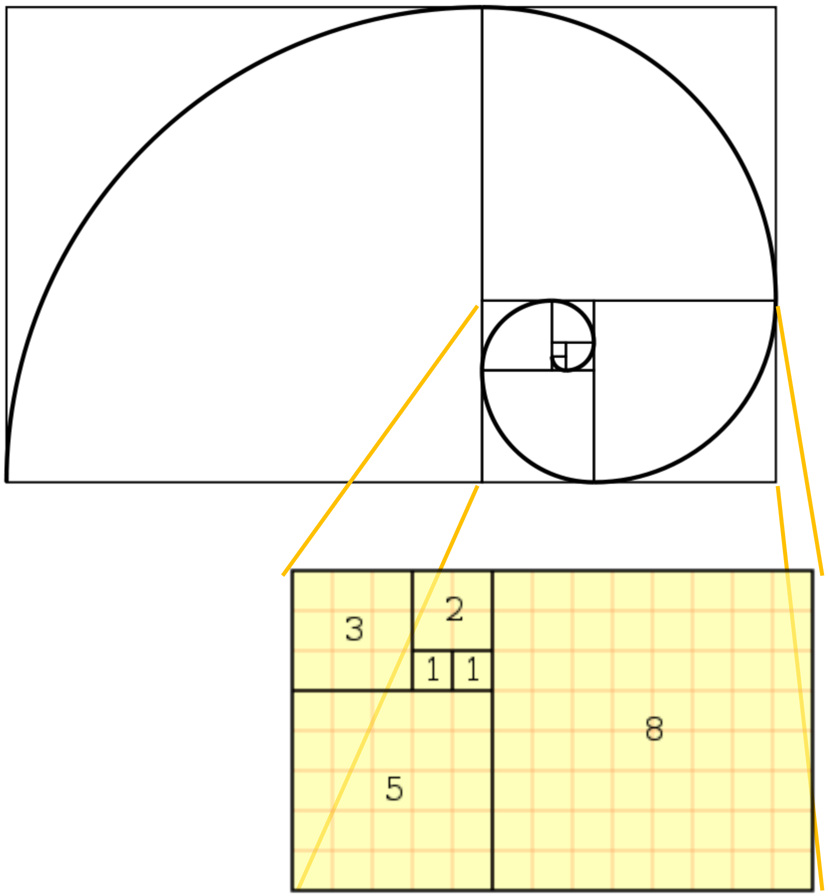
Divide-and-Conquer

- Divide and conquer:
  - (Divide) Recursively break down a problem into two or more sub-problems of the same (or related) type
  - (Conquer) Until these become simple enough to be solved directly
  - (Combine) The solutions to the sub-problems are then combined to give a solution to the original problem
- Correctness: proved by mathematical induction
- Complexity: determined by solving recurrence relations

Example: Fibonacci Sequence

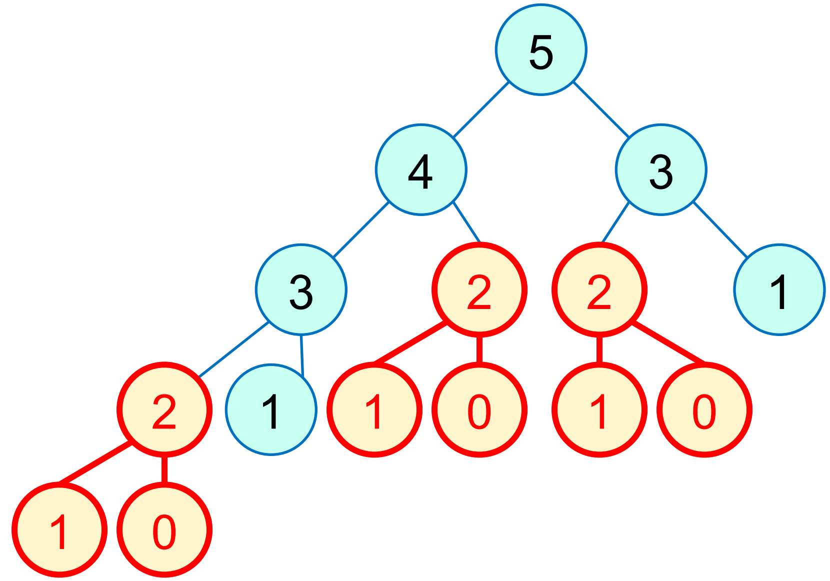
- Recurrence relation: $F_n = F_{n-1} + F_{n-2}$, $F_0=0$, $F_1=1$
  - e.g., 0, 1, 1, 2, 3, 5, 8, …
- Direct implementation:
  - Recursion!

fib(n)
1. **if** n = 0 **return** 0
2. **if** n = 1 **return** 1
3. **return** fib(n − 1) + fib(n − 2)



What's Wrong?

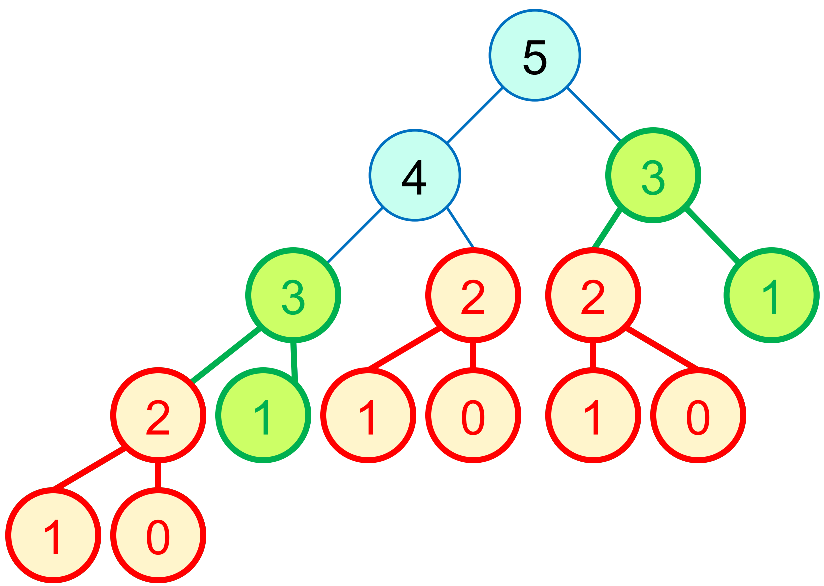
```
fib(n)
if n == 0 return 0
if n == 1 return 1
return fib(n − 1) + fib(n − 2)
```



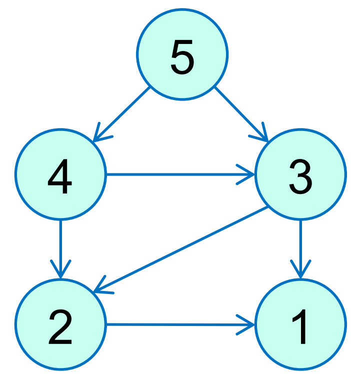
- What if we call fib(5)?
  - fib(5)
  - fib(4) + fib(3)
  - (fib(3) + fib(2)) + (fib(2) + fib(1))
  - ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
  - (((fib(1) + fib(0)) + fib(1))+(fib(1) + fib(0)))+((fib(1) + fib(0))+fib(1))
  - A call tree that calls the function on the same value many different times
    - fib(2) was calculated three times from scratch
    - Impractical for large *n*

Too Many Redundant Calls!

Recursion	True dependency



- How to remove redundancy?
  - Prevent repeated calculation

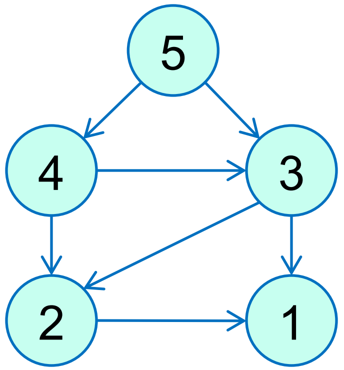


Dynamic Programming

- Store the values in a table
  - Check the table before a recursive call
  - Top-down!
    - The control flow is almost the same as the original one

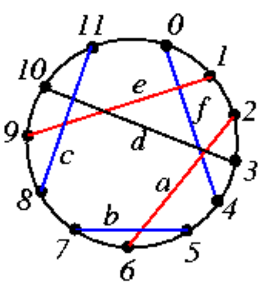
```
fib(n)
1.  Initialize f[0..n] with -1 // -1: unfilled
2.  f[0] = 0; f[1] = 1
3.  fibonacci(n, f)

fibonacci(n, f)
1.  If f[n] == -1 then
2.     f[n] = fibonacci(n − 1, f) + fibonacci(n − 2, f)
3.  return f[n] // if f[n] already exists, directly return
```

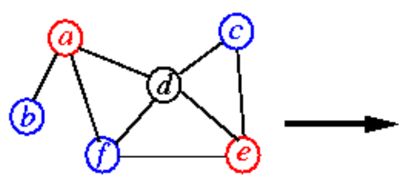


Maximum Independent Set (MIS) in Circle Graphs

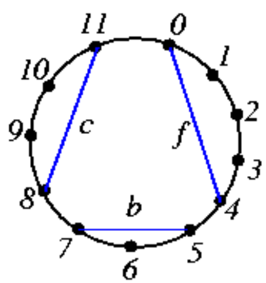
- MIS in general is NP-complete
- Problem: Given a set of chords, find a maximum planar subset of chords.
  - Label the vertices on the circle 0 to $2n$-1.
  - Compute $MIS(i, j)$: size of MIS between vertices i and j, $i < j$.
  - $MIS(0, 2n\text{-}1)$ is efficiently solvable by dynamic programming.



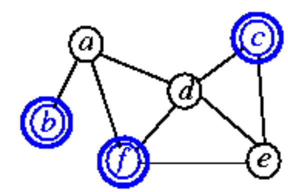
A set of chords.



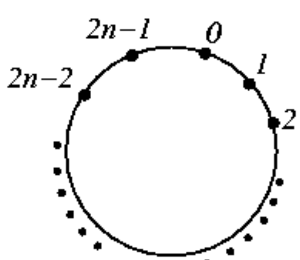
circle graph



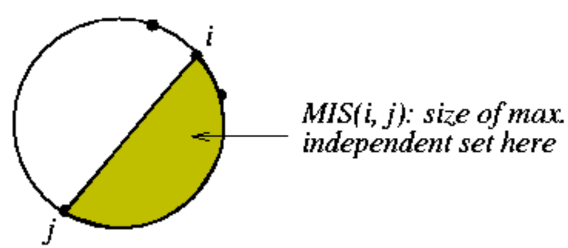
Maximum plannar subset of chords.



Maximum independent set: nodes b, c, f



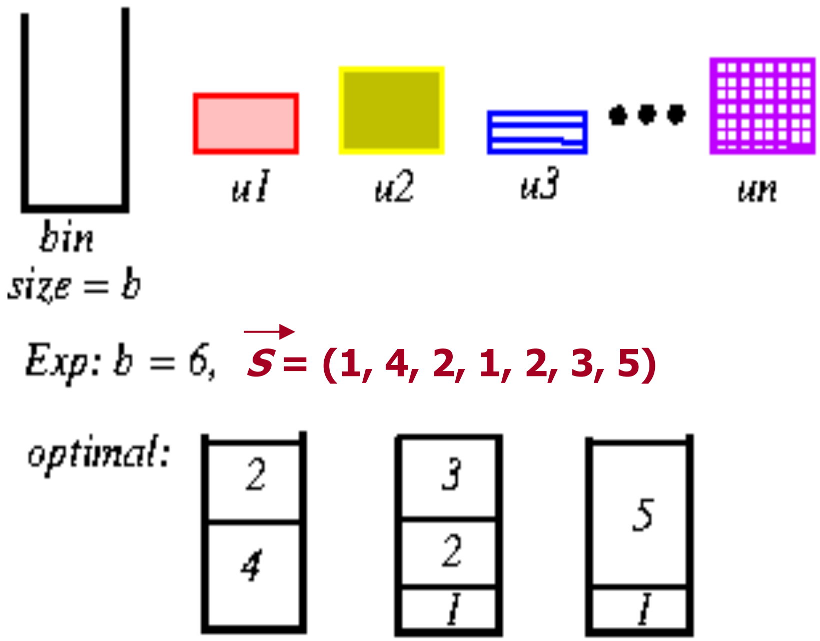
1 **Vertices on the circle**

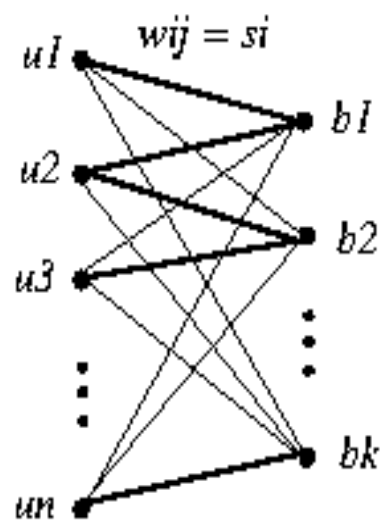


$MIS(i, j)$, $i < j$

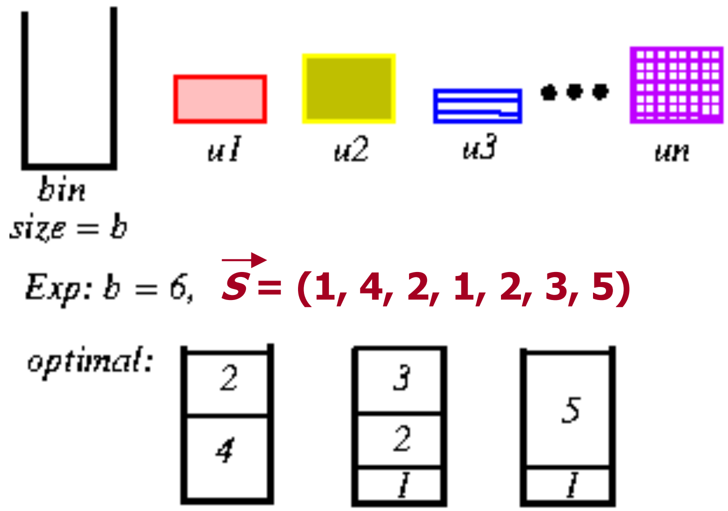
Example: Bin Packing

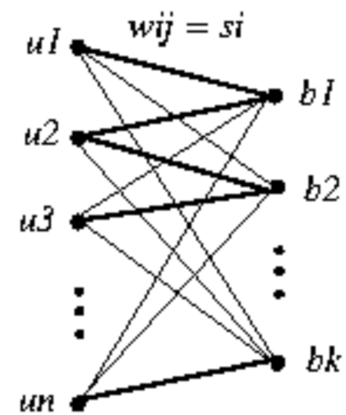
- **The Bin-Packing Problem** Π **:** Items $U = \{u_1, u_2, \ldots, u_n\}$, where u_i is of an integer size s_i; set B of bins, each with capacity b.
- **Goal:** Pack all items, minimizing # of bins used. (**NP-hard!**)





Algorithms for Bin Packing





- Greedy approximation alg.: First-Fit Decreasing (FFD)
  - $FFD(\Pi) \leq 11OPT(\Pi)/9 + 4$
- Dynamic Programming? Hierarchical Approach? Genetic Algorithm? …
- Mathematical Programming: Use **integer linear programming (ILP)** to find a solution using $|B|$ bins, then search for the smallest feasible $|B|$.

ILP Formulation for Bin Packing

- 0-1 variable: x_{ij}=1 if item u_i is placed in bin b_j, 0 otherwise.

objective function

$$\max \sum_{(i,j)\in E} w_{ij}x_{ij}$$

subject to

constraints

$$\begin{aligned}
\sum_{\forall i\in U} w_{ij}x_{ij} &\leq b_j, \forall j \in B \quad /*\ \textit{capacity constraint}\ */ && (1)\\
\sum_{\forall j\in B} x_{ij} &= 1, \forall i \in U \quad /*\ \textit{assignment constraint}\ */ && (2)\\
\sum_{ij} x_{ij} &= n \quad /*\ \textit{completeness constraint}\ */ && (3)\\
x_{ij} &\in \{0,1\} \quad /*\ 0,\ 1\ \textit{constraint}\ */ && (4)
\end{aligned}$$

- **Step 1:** Set $|B|$ to the lower bound of the # of bins.
- **Step 2:** Use the ILP to find a feasible solution.
- **Step 3:** If the solution exists, the # of bins required is $|B|$. Then exit.
- **Step 4:** Otherwise, set $|B| \leftarrow |B| + 1$. Goto Step 2.

Machine Learning for EDA

Problem types solved with Machine Learning

- Classification
- Regression
- Dimensionality reduction
- Structured prediction
- Anomaly detection

Past ML applications in EDA literature

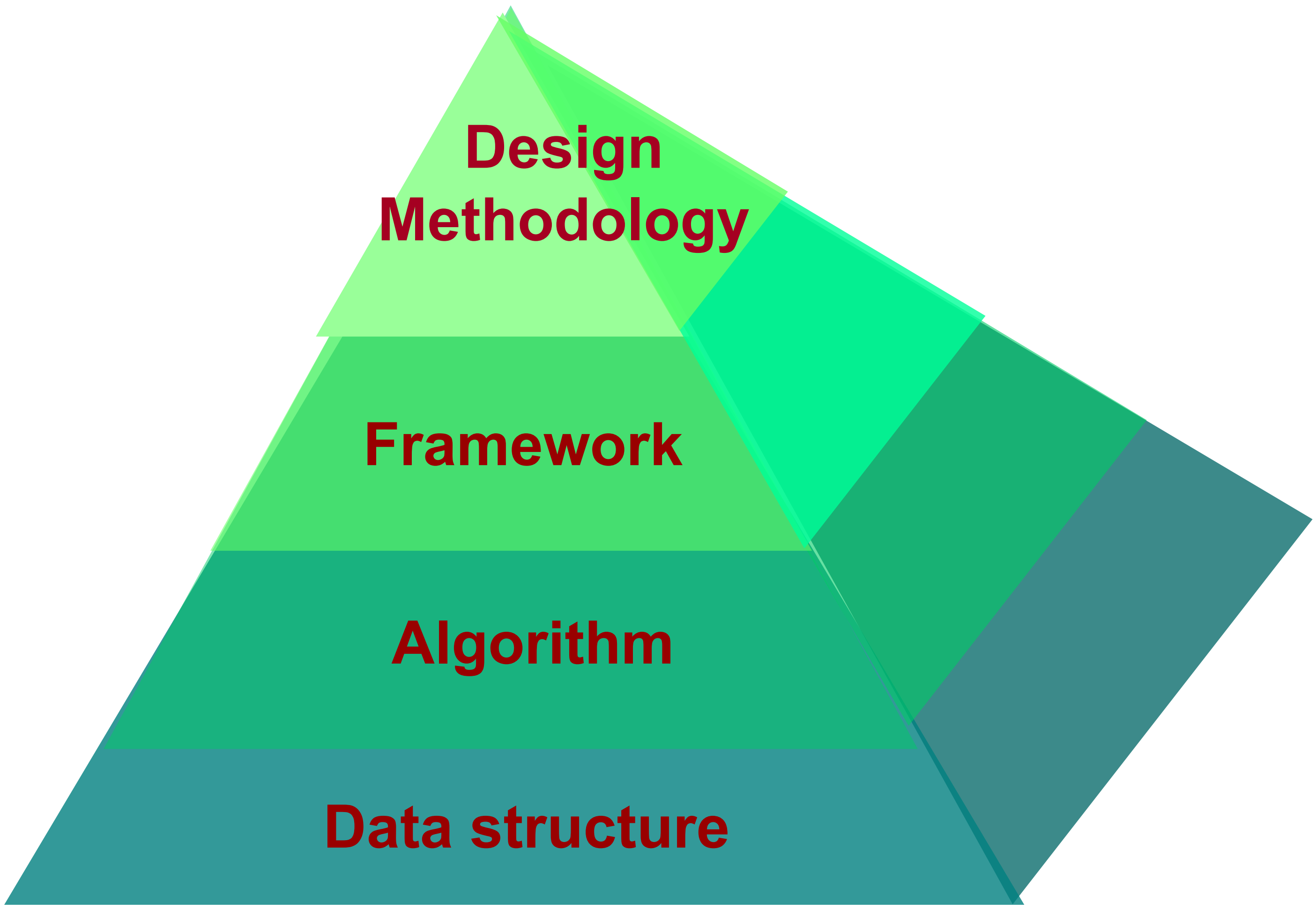
- Yield modeling (anomaly detection, classification)
- Lithography hotspot detection (classification)
- Identification of datapath-regularity (classification)
- Noise and process-variation modeling (regression)
- Performance modeling for analog circuits (regression)
- Design- and implementation-space exploration (regression)

ML in PD: modeling, prediction, correlation, …

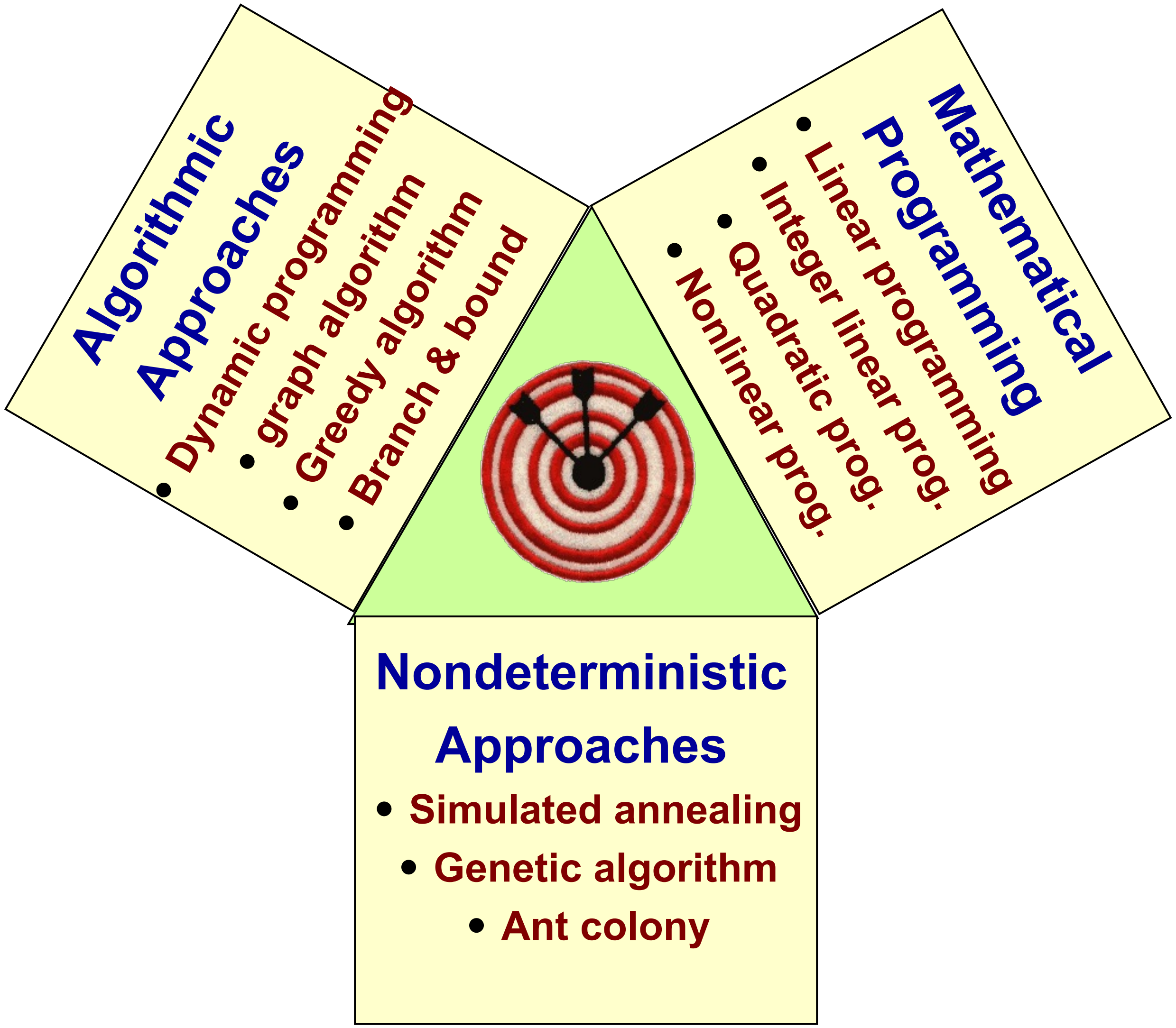
UCSD

A. B. Kahng, 180327 ISPD--2018

Pyramid for Solving an EDA Problem



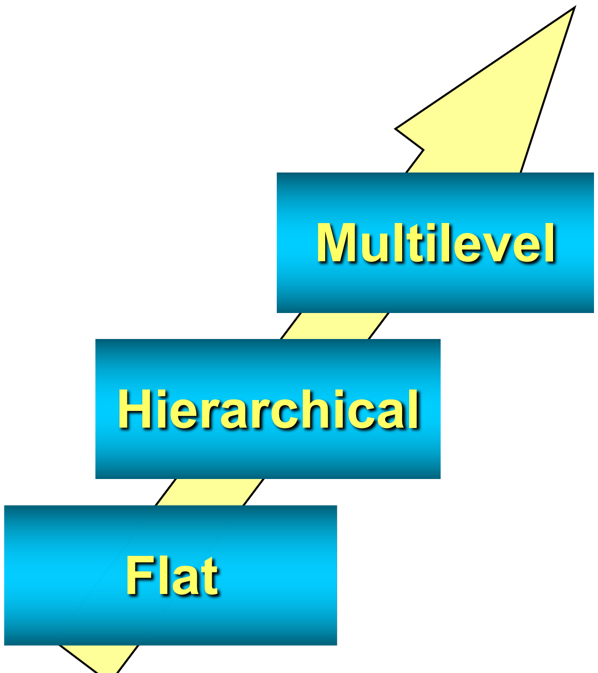
Classifications of Popular EDA Algorithms

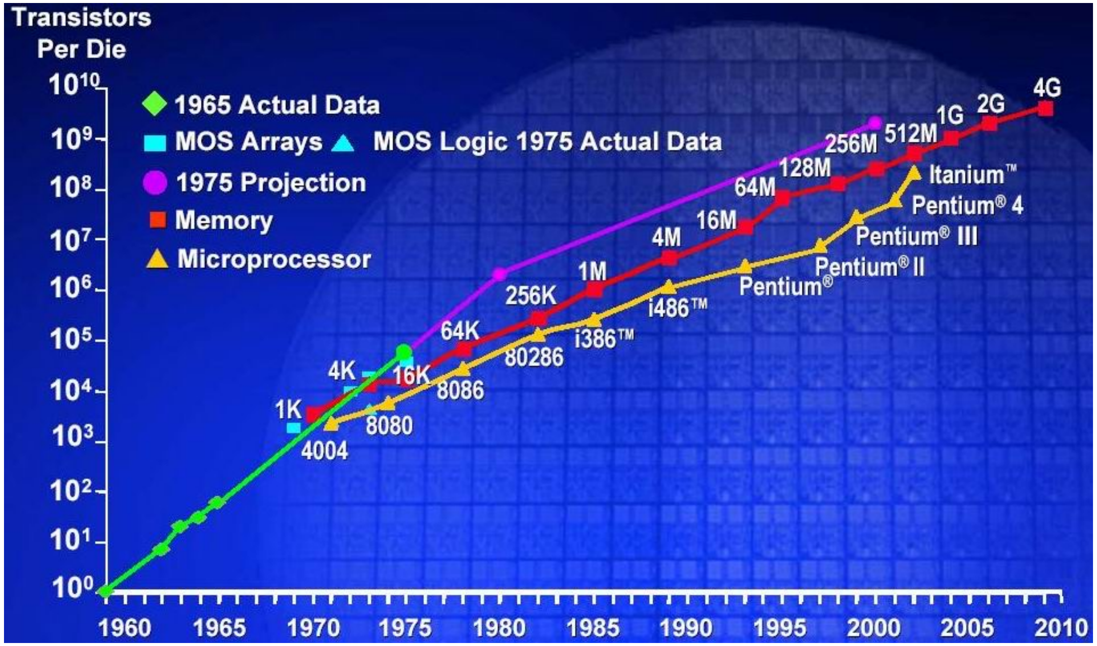


Framework Evolution

- Billions of transistors may be fabricated in a single chip for nanometer technology.
- Need frameworks for very large-scale designs.
- Framework evolution for EDA tools:

Flat ➔ Hierarchical ➔ Multilevel

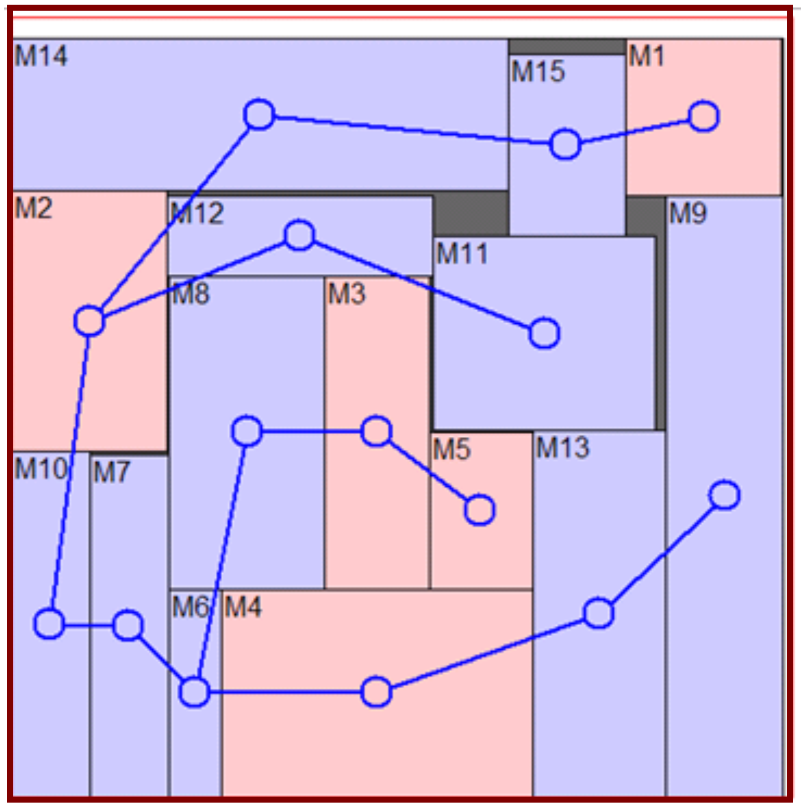




Source: Intel (ISSCC-03)

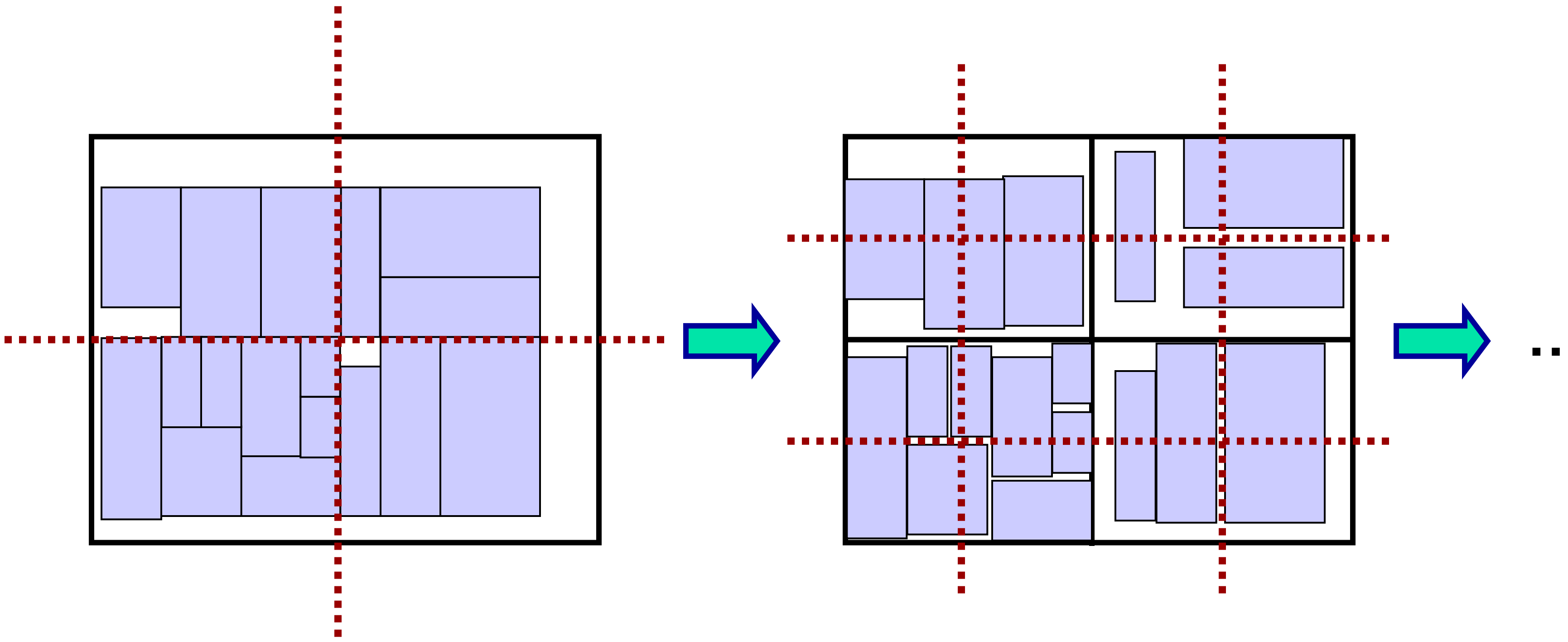
Flat Framework for 2D Bin Packing (Floorplanning)

- Process the circuit components in the whole chip
- Limitation: Good for small-scale designs, but hard to handle larger problems directly



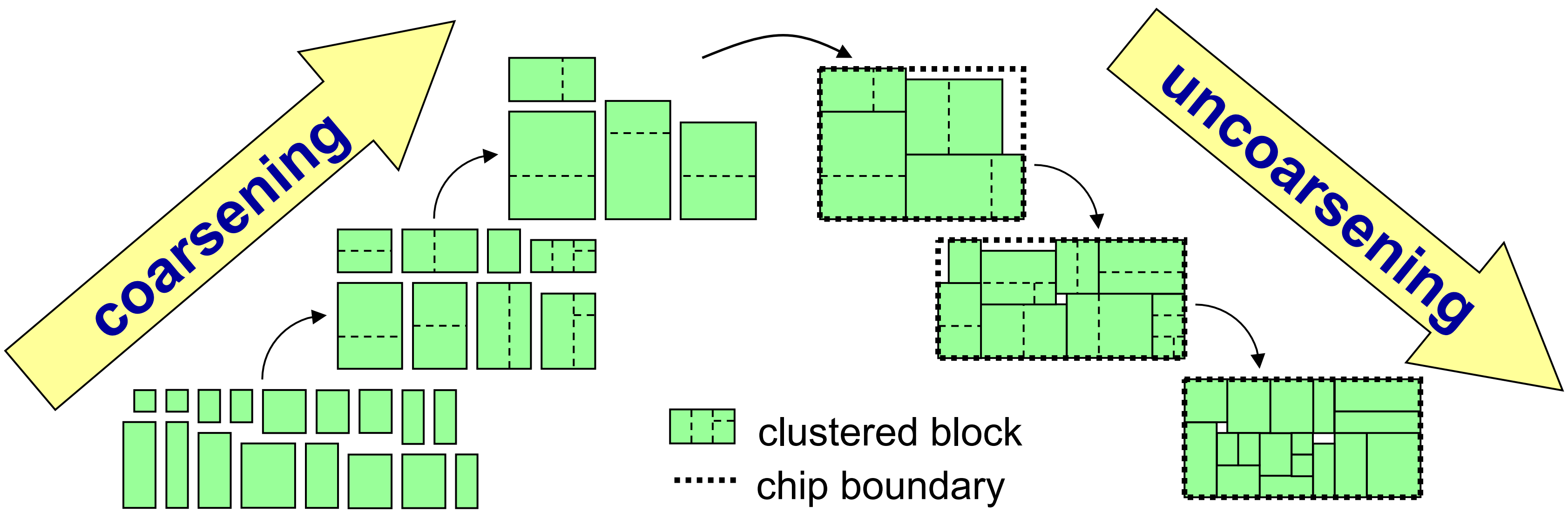
Hierarchical Framework

- The hierarchical approach recursively divides a circuit region into a set of subregions and solve those subproblems ***independently***.
- Good for scalability for large-scale design, but lack the global information for the interaction among subregions.

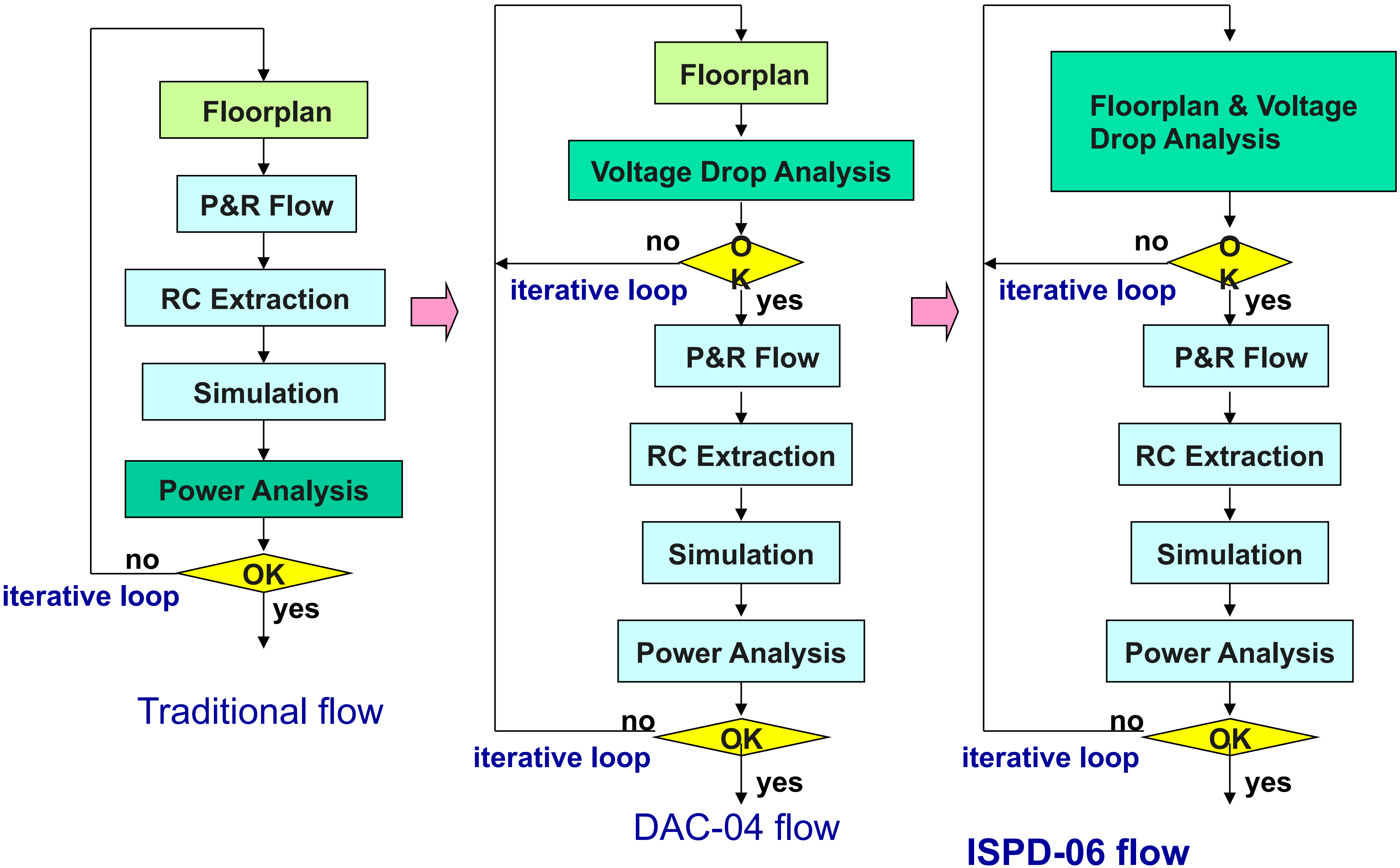


Multilevel Floorplanning

- Bottom-up Coarsening (clustering): Iteratively groups a set of circuit components and collects global information.
- Top-down Uncoarsening (declustering): Iteratively ungroups clustered components and refines the solution.
- Good for scalability and quality trade-off

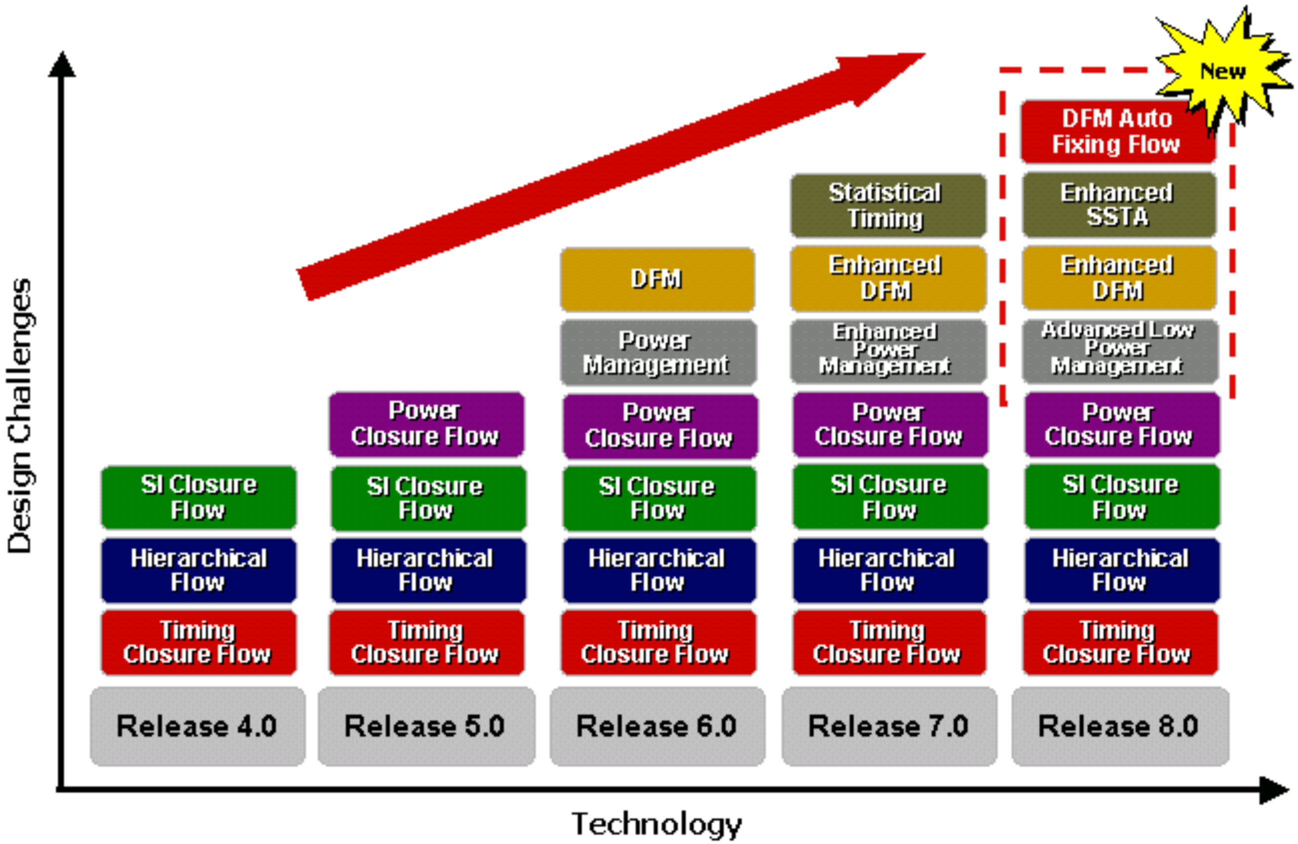


Design Methodology Evolution



TSMC Reference Flow

Reference Flow 8.0
The nanometer design flow for SoC



Physical Design Related Conferences/Journals

- Important Conferences:
  - **ACM/IEEE Design Automation Conference (DAC)**
  - **IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD)**
  - **ACM Int'l Symposium on Physical Design (ISPD)**
  - ACM/IEEE Asia and South Pacific Design Automation Conf. (ASP-DAC)
  - ACM/IEEE Design, Automation, and Test in Europe (DATE)
  - IEEE Int'l Conference on Computer Design (ICCD)
  - IEEE Int'l Symposium on Circuits and Systems (ISCAS)
  - IEEE-TSA VLSI Design, Automation and Test (VLSI-DAT)
  - Many more, e.g., GLSVLSI, ISLPED, ISQED, SOCC, VLSI, VLSI Design/CAD Symposium/Taiwan
- Important Journals:
  - **IEEE Transactions on Computer-Aided Design (TCAD)**
  - **ACM Transactions on Design Automation of Electronic Systems (TODAES)**
  - **IEEE Transactions on VLSI Systems (TVLSI)**
  - **IEEE Transactions on Computers (TC)**
  - INTEGRATION: The VLSI Journal
  - IEE Proceedings