The sphere of algorithmic problems
- **Traveling Salesman Problem (TSP):** find the shortest route that passes through each of the nodes in a graph exactly once and returns to the starting node.

```
E ——— 5 ——— A ——— 2 ——— B
   ^          |          ^
   |          |          | 2
     3        1
D ——— 4 ——— C ——— 3
     2

Shortest route: ABDECA
Length(ABDECA) = 2 + 2 + 2 + 2 + 3 = 11
```
Exhaustive search: compute all paths and select the one with the minimum cost

- Length(ABCDEA) = 2 + 3 + 6 + 2 + 5 = 18
- Length(ACBDEA) = 3 + 3 + 2 + 2 + 5 = 15

......

- Length(ABDECA) = 2 + 2 + 2 + 2 + 3 = 11 → BEST PATH
- Length(AEBCDA) = 5 + 1 + 3 + 6 + 4 = 19

(n-1)! paths ➔ Complexity: $\Omega(2^n)$
- **Hard Problems**: an exponential algorithm that solves the problem is known to exist
  - E.g., TSP
- Is there a better algorithm?
  - Until when do we try to find a better algorithm?
- Prove that the problem as at least as hard as another hard problem for which no better solution has even been found
  - Then, stop searching for a better solution for the first problem
NP Complete problems (NPC): hard problems for which only exponential algorithms are known to exist

- Polynomial solutions might exist but none has found yet!
- Examples:
  - Traveling Salesman Problem (TSP)
  - Hamiltonian Path Problem
  - 0-1 Knapsack Problem

Properties of NP Completeness:
- Non deterministic polynomial
- Completeness
Non-deterministic polynomial: a polynomial algorithm doesn't guarantee optimality

- The polynomial algorithm is non-deterministic: tries to find a solution using heuristics
- E.g., TSP: the next node is the one closer to the current node

Starting point

Length(ABEDCA) = 2 + 1 + 2 + 6 + 3 = 14
Non optimal
Completeness: if an optimal algorithm exists for one of the them, then a polynomial algorithm can be found for all.

- It is possible to transform each problem to another using a polynomial algorithm.
- The complexity for solving a different problem is the complexity of transforming the problem to the original one (takes polynomial time) plus the complexity of solving the original problem (take polynomial time again)!
- **Hamiltonian Path (HP) problem**: is there a path that visits each node of a graph exactly once?
  - The **exhaustive search** algorithm checks $n!$ paths
  - **HP is NP Complete**
  - It is easy to transform HP to TSP in polynomial time
  - Create a **full graph $G'$** having cost 1 in edges that exist in $G$ and cost 2 in edges that don't belong to $G$
The transformation of HP to TSP takes polynomial time

- Cost of creating $G$ + cost of adding extra edges: $O(n)$

- HP: equivalent to searching for a TSP path on $G'$ with length $n + 1$
Algorithms

- Types of Heuristic algorithms
  - Greedy
  - Local search
- Types of Optimal algorithms
  - Exhaustive Search (ES)
  - Divide and Conquer (D&C)
  - Branch and Bound (B&B)
  - Dynamic Programming (DP)
Greedy Algorithms: whenever they make a selection they choose to increase the cost by the minimum amount.

E.g., TSP: the next node is the one closer to the current node.

Greedy algorithm:
Length(BECADB) = 1 + 2 + 3 + 4 + 2 = 12
Non optimal

Optimal: Length(BACEDB) = 2 + 3 + 2 + 2 + 2 = 11
In some cases, greedy algorithms find the optimal solution

- **Knapsack problem**: choosing among \( n \) objects with different values and equal size, fill a knapsack of capacity \( C \) with objects so that the value in the knapsack is maximum
  - Fill the knapsack with the most valuable objects
- **Job-scheduling**: which is the service order of \( n \) customers that minimizes the average waiting time?
  - Serve customers with the less service times first
- **Minimum cost spanning tree**: next transparency
Minimum Cost Spanning Tree (MCST)

- In an undirected graph $G$ with costs, find the set of edges that has minimum total cost and keeps all nodes connected
  - **Application**: connect cities by telephone in a way that requires the minimum amount of wire
  - **MCST contains no cycles (it's a tree)**
  - Exhaustive search takes exponential time (choose among $n^{n-2}$ trees or among $n!$ edges)
  - Two standard algorithms (next transparency)
- **Prim's algorithm**: at each step take the minimum cost edge and connect it to the tree.
- **Kruskal's algorithm**: at each step take the minimum cost edge that creates no cycles in the tree.
- **Both algorithms are optimal!!**
Prim's algorithm:

graph \( G=(V,E), V=\{1,2,\ldots,n\} \)

function \( \text{Prim}(G:\text{graph}, \text{MST}: \text{set of edges}) \)
\( U: \text{set of edges}; u, v: \text{vertices}; \)
{
  \( T = 0; \ U = \{1\}; \)
  \( \text{while } (U \neq V) \{ \)
  \( (u,v) = \text{min. cost edge: } u \text{ in } U, v \text{ in } V \)
  \( T = T + \{(u,v)\}; \)
  \( U = U + \{v\}; \)
  \}
  \}

\textit{Complexity: } \( O(n^2) \) why??

The tree contains \( n - 1 \) edges
Local Search

- Heuristic algorithms that improve a non-optimal solution
- Local transformation that improves a solution
  - E.g., a solution obtained by a greedy algorithm
- Apply many times and as long as the solution improves
- Apply in difficult (NP problems) like TSP
  - E.g., apply a greedy algorithm to obtain a solution, improve this solution using local search
- The complexity of the transformation must be polynomial
Local search on TSP:

Starting point: greedy

Cost = 11: optimal

Cost = 12

Local cost = 5

Local cost = 4
Divide and Conquer

- The problem is split into smaller sub-problems
- Solve the sub-problems
- Combine their solutions to obtain the solution of the original problem
- The smaller a sub-problem is, the easier it is to solve it
- Try to get sub-problems of equal size
- D&C is often expressed by recursive algorithms
  - E.g. Mergesort
Merge sort

```plaintext
list mergesort(list L, int n)
{
    if (n == 1) return (L);
    L1 = lower half of L;
    L2 = upper half of L;
    return merge (mergesort(L1,n/2), mergesort(L2,n/2));
}
```

- **n**: size of array L (assume L some power of 2)
- **merge**: merges the sorted L₁, L₂ in a sorted array
Complexity: $O(n \log n)$
Dynamic Programming

- The original problem is split into smaller sub-problems
  - Solve the sub-problems
  - Store their solutions
  - Reuse these solutions several times when the same partial result is needed more than once in solving the main problem
  - DP is often based on a recursive formula for solving larger problems in terms of smaller
  - Similar to Divide and Conquer but recursion in D&C doesn’t reuse partial solutions
### DP example (1) Fibonacci numbers

\[
F(n) = \begin{cases} 
F(0) = 0 \\
F(1) = 0 \\
F(n) = F(n-1) + F(n-2) & \text{if } n \geq 2
\end{cases}
\]

- **Using recursion the complexity is**
  \[F(n) \in \Theta(\phi^n)\]

- **Using a DP table the complexity is**
  \[O(n)\]
0-1 Knapsack

- 0-1 knapsack problem: given a set of objects that vary in size and value, what is maximum value that can be carried in a knapsack of capacity $C$?
- How should we fill-up the capacity in order to achieve the maximum value?
Exhaustive (1)

- The obvious method to solve the 0-1 knapsack problem is by trying all possible combinations of \( n \) objects.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- Each object corresponds to a cell.
- If it is included its cell becomes 1.
- If it is left out its cell becomes 0.
- There are \( 2^n \) different combinations.
Notation

- \( n \) objects
- \( s_1, s_2, s_3, \ldots, s_n \): capacities
- \( v_1, v_2, v_3, \ldots, v_n \): values
- \( C \): knapsack capacity
- Let \( 0 \leq i \leq n \) and \( A \leq C \)
- \( V(k, A) \): maximum value that can be carried in a knapsack of capacity \( A \) given that we choose its contents from among the first \( k \) objects
- \( V(n, C) \): maximum value that can be carried in the original knapsack when we choose from among all objects
- \( V(k, A) = 0 \) if \( k = 0 \) or \( A \leq 0 \) for any \( k \)
DP Formulation

- \( V(k, A) = \max\{V(k-1, A), V(k-1, A-s_k) + v_k\} \): solving the knapsack problem for the next object \( k \), there are two choices:
  - We can include it or leave it out
  - If it is left out, we can do better by choosing from among the previous \( k-1 \) objects
  - If it is included, its value \( v_k \) is added but the capacity of the knapsack is reduced by the capacity \( s_k \) of the \( k \)-th object
Exhaustive (2)

- If we try to compute $V(n,c)$ by recursive substitution the complexity is $\Omega(2^n)$
  - Two values of $V(n-1,A)$ must be determined for different $A$,
  - Four values of $V(n-2,A)$ and so on
  - Number of partial solutions: number of nodes in a full binary tree of depth $n$: $\Omega(2^n)$
- The number of values to be determined doubles at each step
Dynamic Programming

- Many of these values are likely to be the same especially when $C$ is small compared with $2^n$
- DP computes and stores these values in a table of $n+1$ rows and $C$ columns
**Dynamic Programming (DP):** compute and store $V(0,A)$, $V(1,A)$, ... $V(k,A)$, ... $V(n,C)$ where $0 \leq k \leq n$ and $0 \leq A \leq C$ using $V(k,A) = \max\{V(k-1,A), V(k-1,A-s_k)+v_k\}$.
DP on Knapsack

- Each partial solution $V(k,A)$ takes constant time to compute.
- The entire table is filled with values.
- The total time to fill the table is proportional to its size $\Rightarrow$ the complexity of the algorithm is $O(nC)$.
  - Faster than exhaustive search if $C \ll 2^n$.
- Which objects are included?
  - Keep this information in a second table $X_i(k,A)$.
  - $X_i(k,A) = 1$ if object $i$ is included, $0$ otherwise.
Shortest Path

- find the shortest path from a given node to every other node in a graph $G$
- No better algorithm for single ending node
- Notation:
  - $G = (V,E)$: input graph
  - $C[i,j]$: distance between nodes $i$, $j$
  - $V$: starting node
  - $S$: set of nodes for which the shortest path from $v$ has been computed
  - $D(W)$: length of shortest path from $v$ to $w$ passing through nodes in $S$
starting point: \( v = 1 \)

<table>
<thead>
<tr>
<th>step</th>
<th>( S )</th>
<th>( W )</th>
<th>( D(2) )</th>
<th>( D(3) )</th>
<th>( D(4) )</th>
<th>( D(5) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1}</td>
<td>-</td>
<td>10</td>
<td>( \infty )</td>
<td>30</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>{1,2}</td>
<td>2</td>
<td>10</td>
<td>60</td>
<td>30</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>{1,2,4}</td>
<td>4</td>
<td>10</td>
<td>50</td>
<td>30</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>{1,2,4,3}</td>
<td>3</td>
<td>10</td>
<td>50</td>
<td>30</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>{1,2,4,3,5}</td>
<td>5</td>
<td>10</td>
<td>50</td>
<td>30</td>
<td>60</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm

- function Dijkstra(G: graph, int v)
  
  ```
  S = {1};
  for i = 2 to n:  D[i] = C[i,j];
  while (S != V) {
    choose w from V-S: D[w] = minimum
    S = S + {w};
    for each v in V–S: D[v] = min{D[v], D[w]+[w,v]}*;
  }
  ```

- Complexity: $O(n^2)$

DP on TSP

- The standard algorithm is $\Omega((n-1)!)$
- If the same partial paths are used many times then DP can be faster!

Notation:
- Number the cities 1, 2, ..., n,
- 1 is the starting-ending point
- $d(i, j)$: distance from node $i$ to $j$
- $D(b, S)$: length of shortest path that starts at $b$ visits all cities in $S$ in some order and ends at node 1
- $S$ is subset of $\{1, 2, ..., n\} - \{1, b\}$
DP Solution on TSP

- TSP needs to compute $D(1,\{2,\ldots,n\})$ and the order of nodes.
- DP computes and stores $D(b, S)$ for all $b$'s.

$$D(b, S) = \min_{a \in S} \{ d(b, a) + D(a, S - \{a\}) \}$$

- TSP needs to compute all $D(b, S)$ starting from empty $S$ and proceeding until $S = \{2, 3, \ldots, n\}$.
- Assume that we start from $b=1$.
- $D(b, \varnothing) = d(b, 1)$.
DP Solution on TSP (con’t)

\[ D(b, S) = \min_{a \in S} \{d(b, a) + D(a, S - \{a\})\} \]

- For each one of the \( n \) possible \( b \)'s there are at most \( 2^{n-1} \) paths
- \( S \) is a subset of a set of size \( n-1 \)
- There are \( n \) values of \( b \) in \( S \)
- \( n2^{n-1} \) values of \( D(b, S) \)
- For each \( D(b, S) \) choose the min. from \( n \) a’s
- The complexity is \( O(n^22^{n-1}) \): still exponential but, better than exhaustive search!
- \( D(a, S-\{a\}) \) are not recomputed (they are stored)
Branch and Bound

- Explores all solutions using constraints (bounds)
  - Lower Bound: min. possible value of solution
  - Upper Bound: max. possible value of solution
- The problem is split into sub-problems
- Each sub-problems is expanded until a solution is obtained as long as its cost doesn’t exceed the bounds
  - Its cost must be greater than the lower bound
  - and lower than the upper bound
Upper-Lower Bounds

- The upper bound can be set to $\infty$ initially
  - Takes the cost of the first complete solution as soon as one is found
  - A greedy algorithm can provide the initial upper bound (e.g., in TSP, SP etc)
  - It is revised in later steps if better solutions are found
- The lower bound is not always easy to compute
  - Depends on the problem
  - There is a theorem for TSP
upper bound by greedy:
length(saefkt)=8
Basic Branch and Bound Algorithm

\[ S = \{ 1, 2, \ldots, n \} \] /* initial problem */
\[ U = \infty \] /* upper bound */

while \( S \neq \text{empty} \) {
    \( K = k \in S, S = S - \{k\} \);
    \( C = \{ c_1, c_2, \ldots, c_m \} \) /* children of \( k \) */
    Computes partial solutions \( Zc_1, Zc_2, \ldots, Zc_m \)
    for \( i = 1 \) to \( m \) {
        if \( Zc_i \geq U \) kill \( C \)
        else if ( \( c_i = \text{solution} \) ) {
            \( U = Zc_i \)
            best solution = \( i \)
        } else
            \( S = S + \{ c_i \} \)
    }
}