Metod 2: Giriga algoritmer (greedy algorithms)

Algoritmer som löser en bit av problemet i taget.

I varje steg görs det som ger bäst utdelning / kostar minst.

Exempel 1: kortaste stig med Dijkstras algoritm:

En mängd S med hörn till vilka kortaste stigen är känd utvidgas girigt:

"Utvidga S med det hörn som är märkt med det kortaste avståndet och uppdatera märkningen"

Exempel 2: Minimalt spännande träd med Prims algoritm:

Ett träd utvidgas girigt tills det spänner hela grafen:

"Utvidga trädet med det hörn i grafen som har kortast avstånd till något hörn i trädet"

I dessa fall hittar dom giriga algoritmerna optimala lösningar, men för många andra problem ger giriga algoritmer inte optimala lösningar utan kanske hyfsade approximativa lösningar.

Metod 3: Totalsökning (exhaustive search)

- Gå igenom alla tänkbara lösningar och kolla om det är den sökta lösningen

  Detta görs med fördel rekursivt.

Ofta är antalet tänkbara lösningar exponentiellt många (tex 2^n) och då går det bara att använda för små n. Totalsökning är en metod man tar till i sista hand.

Det svåraste med totalsökning är normalt att se till att man går igenom varje tänkbar lösning en (och helst inte mer än en) gång. Att sedan kolla om det är den sökta (eller optimala) lösningen brukar vara lätt.

Ibland kan man redan innan en tänkbar lösning är färdigkonstruerad se att den inte är möjlig som lösning. Då kan man strunta i att gå vidare med den och istället gå tillbaka och konstruera nästa möjliga lösning. Detta kallas backtracking.

Exempel: lösning av det generella handelsresandeproblemet.

TSP – handelsresandeproblemet (traveling salesperson problem)

Hitta kortaste turen som passerar alla städer en gång.

Olika varianter av problemet:

- Generell TSP

  probleminstans: graf med kantvikter

- Euklidisk TSP i dimension d

  probleminstans: städerna givna som koordinater i $\mathbb{R}^d$

Lösning av generell TSP med totalsökning

Datastrukturer: perm[1..n] här konstrueras vi varje tänkbar lösning, dvs permutation av städerna
visited[1..n] anger om en stad besöks hittills i den permutation som konstrueras

```
TSP(n, d[1..n,1..n]) =
  minlen ← ∞;
  FOR i ← 1 TO n DO visited[i] ← FALSE;
  FOR i ← 1 TO n DO
    perm[1] ← i; visited[i] ← TRUE;
    CheckPerm(2, 0);
    visited[i] ← FALSE;
  RETURN minlen

CheckPerm(k, length) =
  IF k > n THEN
    totalLength ← length + d[perm[n], perm[1]];
    IF totalLength < minlen THEN minlen ← totalLength;
  ELSE ← backtracking kan införas här med satsen IF length < minlen THEN
    FOR i ← 1 TO n DO
      IF NOT visited[i] THEN
        perm[k] ← i; visited[i] ← TRUE;
        CheckPerm(k+1, length + d[perm[k-1], i]);
        visited[i] ← FALSE;
```

Tidskomplexitet: $O(n^2 \cdot n!)$ vilket enkelt kan minskas till $O(n!)$ om man håller reda på vilka städer som ännu inte besöks effektivare (t ex med en kö)

8-damsproblemet

Placera ut 8 damer på ett schackbräde utan att någon pjäs hotar någon annan!

Totalsökningsalgoritm:

Placering av en dam på rad row:

- Pröva att placera damen i varje position på raden i tur och ordning
- Kan damen stå ohotad i den positionen så placeras nästa dam på rad row+1 ut på samma sätt, om inte row=8 för då har man hittat en lösning.

```
int queenpos[9];

void TestRow(int row)
{ int i;
  for (i = 1; i<=8; i++) {
    queenpos[row] = i;
    if (PosOK(row)) {
      if (row == 8) WriteSolution();
      else TestRow(row+1);
  } }
}
```

Starta med TestRow(1);