SEARCH

ในเรื่องของการ Search นั้น ลำดับของการ Search มีความสำคัญ ซึ่งจะเห็นได้ในเรื่องของ Depth-first search และ Breadth-first search ต่อไป

การ Search โดย DFS หรือ BFS ธรรมดาตรงๆ นั้น อาจทำให้จำนวน node ที่ถูกสร้างขึ้นมามีจำนวนมากเกินความจำเป็น และมักโตเป็น exponential อีกด้วย ดังนั้น เราต้องพยายามลดจำนวน node ลง โดยจะพูดถึงในเรื่องของ Backtracking และ Branch & Bound

Depth-first search และ Breadth-first search

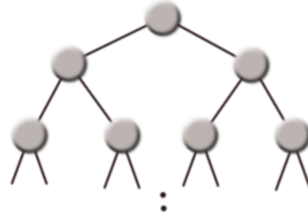
เราเคยเรียนมาใน class ก่อนหน้านี้แล้วว่า ลักษณะของการ search ก็คือ การเข้าไปหยิบค่าใน Storage ออกมาตรวจสอบ แล้วทำการ generate ค่าใหม่ลงไปเก็บใน Storage ซึ่งมี Pseudo code ของ Framework ดังนี้

```
// Idea of Search
Storage S
      S ← ' '                                       // เก็บข้อมูลของ Step ที่ยังไม่ได้ทำอะไร ลงไปใน Storage
      While S is not empty {
            Curr = S.get                            // เอาค่าออกมาจาก Storage
            If Curr is last step
                  Evaluate                          // เอา Solution มาตรวจสอบ
            Else
                  Generate all next step from Curr
                  Put them into S
```

จาก Pseudocode ด้านบน เรายังไม่สามารถเขียน Search tree ได้ เนื่องจาก เราไม่รู้ลักษณะการโตของ tree (ใครเกิดก่อน ใครเกิดหลัง ใครสร้างใคร) สิ่งหนึ่งที่จะบ่งบอกลักษณะการโตของ tree ได้ก็คือ Storage

ถ้าเราเลือกใช้ Stack เป็น Storage การทำงานของโค้ดข้างบนจะเป็นแบบ Depth-first search (DFS) การโตของ tree ก็จะออกไปตามแนวลึก ดังภาพ

ถ้าเราเลือกใช้ Queue เป็น Storage การทำงานของโค้ดก็จะเป็นแบบ Breadth-first search (BFS) การโตของ tree จะออกไปเป็นแนวกว้าง คือ ทีละ level



ถ้าเลือกใช้การ Recursive แทนการมี Storage เก็บข้อมูล ก็จะมีลักษณะการทำงานเหมือนใช้ Stack

ลองมาดูตัวอย่างที่ใช้การ Search ในการแก้ปัญหา

ปัญหาคูณสามหารสอง

Input : จำนวนเต็ม 1 จำนวน

Output : Sequence ของการใช้วิธี *3 หรือ /2 ที่ทำให้ได้มาซึ่งจำนวนที่กำหนด โดยเริ่มจาก 1

Search space สำหรับปัญหาข้อนี้ ย่อมเป็นทุก Sequence ที่เป็นไปได้ทั้งหมดของ *3 และ /2 ซึ่งเราสามารถใช้ Array ในการ represent ได้ โดยอาจให้ 0 แทน *3 และ 1 แทน /2 แต่จะพบปัญหาอย่างหนึ่งของ Search space นี้ ก็คือ มันมีได้ไม่จำกัด เนื่องจากไม่มีกำหนดความยาวสูงสุดของ Sequence เอาไว้

ลองดูโค้ดของปัญหาที่นำเอา DFS มาแก้ (ใช้ Stack เป็น Storage)

```
// *3/2 solution1 : normal DFS
void DFS(int goal){
       Stack S;                                    // Store String
       S.push("1");                                // String ของ 1
       while (!S.isEmpty()){
              Curr = S.pop();
              if (test(Curr) == goal){
                     printf("Found!");
                     return;
              }
              char a[10000], b[10000];
              strcpy(a,Curr); strcat(a,"*3");
              S.push(a);
              strcpy(b,Curr); strcat(b,"/2");
              S.push(b);
       }
}
```

จากโค้ดจะเห็นว่า เรามีพื้นที่ในการเก็บ Sequence อย่างจำกัด ก็คือขนาดของ char array ที่ประกาศเอาไว้ ตัวอย่างเช่นในโค้ดนี้ เราใช้ความยาว 10000 แต่ก็ไม่สามารถแน่ใจได้ว่ามันเพียงพอแล้ว รวมทั้งการที่เราสร้างโดยให้แตกลูกออกมาเสมอ ก็อาจจะทำให้ได้ Sequence ที่มีผลลัพธ์ (ที่ไม่ต้องการ) ซ้ำกัน เช่น 1*3*3/2/2/2*3 และ 1*3 ที่สำคัญหากว่าคำตอบ (goal) ที่ต้องการไม่ได้อยู่ใน "กิ่ง" แรกของการทำ DFS จะทำให้การ Search นั้นค้นลึกลงไปเรื่อยๆ อย่างไม่มีที่สิ้นสุด

เราจึงต้องนำโค้ดมาปรับปรุง โดยเราจะให้ Stack เก็บค่า int ที่คำนวณออกมาในแต่ละขั้นแทนการเก็บ String และเพิ่มเงื่อนไขที่ตรวจสอบว่า หากในขั้นตอนใดๆ ที่มีการคำนวณแล้วคำตอบซ้ำ ก็จะไม่เอาขั้นตอนนั้นใส่ลงใน Stack เพื่อป้องกันความซ้ำซ้อนที่เกิดในโค้ดแรก

```
// *3/2 Solution 2 : ถ้าเจอเลขเดิมก็ไม่ทำต่อ
void DFS(int goal){
      Stack S;                                  // Store Integer
      S.push(1);                                // int ค่า 1
      while (!S.isEmpty()){
            Curr = S.pop();
            if (test(Curr) == goal){
                  printf("Found!");
                  return;
            }
            if (!S.contain(Curr*3))             // ถ้ายังไม่มี ค่อยทำ แต่ก็น่าจะเก็บเป็น Set
                  S.push(Curr*3);
            if (!S.contain(Curr/2))             // เพิ่มหน่อยว่า "&& Curr != 0"
                  S.push(Curr/2);
      }
}
```

ลองเขียนโดยใช้วิธี recursive

```
// *3/2 Solution 2 : recursive
void DFS(int goal, int Curr, Hash visited) {
      if(Curr == goal) {
            printf("found"); return;
      }
      else {
            if(!visied.contain(Curr*3))         // มันก็ *3 ไปเรื่อยๆสิ ถ้าไม่เจอ แล้วเมื่อไหร่จะหยุดล่ะ ?
                  DFS(goal, Curr*3, visited);
            if((Curr>1) && !visited.contain(Curr/2))
                  DFS(goal, Curr*3, visited);
      }
}
```

จากทั้งสองวิธีข้างบน จะทำให้ปัญหาการสร้างคำตอบซ้อนหายไป แต่ยังทำให้เราไม่รู้ว่า Sequence ที่ทำให้ได้มาซึ่งคำตอบปัจจุบันคืออะไรอีก รวมทั้งยังคงมีปัญหาเรื่องอาจจะค้นคำตอบไม่เจออีก จึงต้องเปลี่ยนไปใช้วิธีอื่นๆ

ลองนำเอา BFS มาแก้ปัญหาแทน (ใช้ Queue เป็น Storage)

```
// *3/2 Solution 3 : BFS
void BFS(int goal){
        Queue Q;                                        // Store Integer
        Q.enqueue("1");                                 // int ค่า 1
        while (!Q.isEmpty()){
                Curr = Q.dequeue();
                if (test(Curr) == goal){
                        printf("Found!");
                        return;
                }
                if (!Q.contain(Curr*3))
                        Q.enqueue(Curr*3);
                if ((Curr>1) && !Q.contain(Curr/2))
                        Q.enqueue(Curr/2);
        }
}
```

จะเห็นว่าการทำ BFS ซึ่งเป็นการไล่ค้นตามแนวกว้างลงไปทีละ level ทำให้เรามีโอกาสค้นไปได้ในทุกๆกิ่งของ tree ที่เกิดขึ้น และจะมีโอกาสเจอคำตอบเร็วกว่า DFS

แต่ก็มีปัญหา คือ ใน Queue ต้องเก็บ data เท่าไหร่ ก็เท่ากับ จำนวน node ในชั้นที่กำลัง clear อยู่ ซึ่งถ้ากำลังทำชั้นที่ 100 ก็ต้องเก็บข้อมูลถึง 2^{99} ตัว! (Stack DFS ใช้พื้นที่น้อยกว่ามาก)

DFS vs BFS

DFS นั้นเราจะต้องไล่ลงตามแนวลึกแล้วค่อยๆกลับขึ้นมาด้านบน และ ณ เวลาหนึ่งๆ หากเราดูข้อมูลภายใน Stack (หรือข้อมูลที่เกิดจากการวน Recursive) จะพบว่า มีจำนวนข้อมูลเป็น O(2h) เมื่อ h เป็นความสูงของ tree นั่นก็คือเก็บเฉพาะกิ่งหลักที่กำลังค้น และกิ่งย่อยเล็กๆที่แตกออกมาจากกิ่งหลัก 1 ปม ทำให้วิธีนี้ประหยัดเนื้อที่

สำหรับ BFS จะค่อยๆหาคำตอบจาก level บนๆลงมา ทำให้จำนวนข้อมูลใน Queue ณ ขณะหนึ่งๆเท่ากับ $O(2^h)$ เมื่อ h คือความสูงที่กำลังค้น (หรือก็คือ level) ซึ่งกินเนื้อที่มากกว่าการทำ DFS มาก

จากคุณสมบัติข้างต้น สรุปได้ว่า

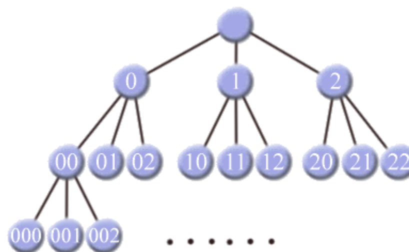
DFS	BFS
ใช้ Stack หรือ Recursive	ใช้ Queue
กินพื้นที่น้อย	กินพื้นที่เยอะ
เหมาะกับการทำ Search ที่ต้องการคำตอบที่เป็นไปได้ทุกรูปแบบ	เหมาะกับการทำ Search ที่ต้องการคำตอบเพียงคำตอบเดียว
อาจจะเจอ Infinite Loop หากไม่ได้จำกัดจำนวน Step	

Backtracking

วิธีนี้จะทำให้เราพบคำตอบเร็วขึ้น โดยมี Idea คือ เราจะไม่แตก node ปัจจุบันที่ไม่มีทางนำไปสู่คำตอบที่ถูกต้อง อย่างกรณีของ 8-queen ตอนแรกเอา Queen วางไว้ตรงไหนก็ได้ แต่มันเกิด Solution ที่ไม่ควรจะเกิด เช่น การวาง Queen ทับกันที่ช่องเดียวกัน หรือการวาง Queen เกิน 8 ตัว เราจึงต้องปรับปรุงโค้ดใหม่ เพื่อตัดปัญหาเหล่านี้ทิ้งไป เช่น อย่าวางตัวที่ 9 เพราะทำต่อยังไงก็ไม่มีทางเจอคำตอบ

Ex1 : Permutation by combination

สมมติว่ามีตัวเลข 0, 1, 2 ให้แสดงการเรียงสับเปลี่ยนทุกๆแบบ ซึ่งถ้าทำด้วยวิธี combination ก็คือเลือกตัวเลขมาทีละตัว



วิธีการทำนั้น เราอาจจะสร้าง combination ทั้งหมดก่อน แล้วค่อยตัดสิ่งที่ไม่ใช่ออก หรือทางที่ดีกว่าก็คือ ตัดขั้นตอนที่ไม่ใช่คำตอบออกไปเลย

จากภาพด้านบน จะเห็นว่ามีกรณีที่ไม่ควรจะแตกลูกต่อด้วย ได้แก่ 00, 11, 22 เพราะว่าการเรียงสับเปลี่ยนของที่มีอยู่ ไม่มีทางเกิดกรณีเหล่านี้ขึ้นมาได้ การที่เราตรวจสอบว่า 00, 11, 22 ไม่สมควรแตกลูกต่อ และไม่ไปทำตรงนั้น ก็คือ วิธี Backtracking นั่นเอง

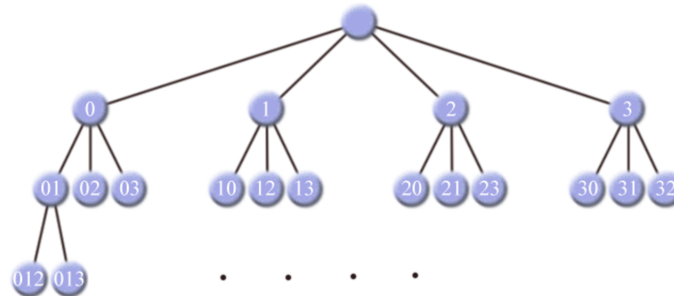
Partial solution

คือ คำตอบใดๆที่เรากำลังสนใจอยู่ในขณะนั้น และกำลังจะถูกนำไปใช้ Generate solution ต่อๆไป ไม่ใช่ Complete solution ที่เราจะนำไปเช็ค (Evaluate) ว่าใช่สิ่งที่ต้องการหรือไม่ อย่างในตัวอย่าง Permutation ข้างบน 0, 1, 2, 00, 11, 22 ก็คือ Partial solution แต่ 012, 102, 210 ไม่ใช่

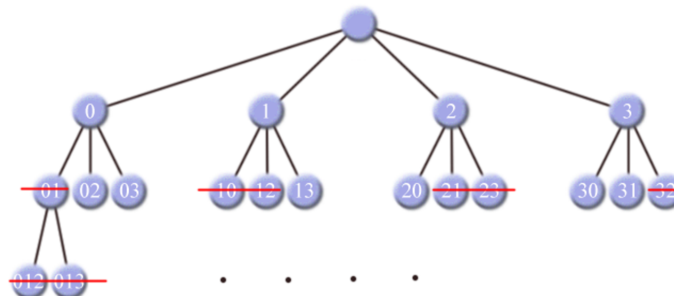
ในการทำ Backtracking and B&B นั้น เราต้องพิจารณา partial solution ก่อน ว่าสมควรไปต่อหรือไม่ ถ้าดู partial solution นั้นแล้ว ไม่มีทางเจอคำตอบได้ ก็ไม่ต้องนำมาทำต่อแล้ว

Ex2 : 4-Queen

ในการวาง Queen รูปแบบต่างๆ ตอน Generate solution ที่ผ่านมาเราใช้วิธีการเลือกวาง Queen 1 ตัวต่อ 1 แถว หากเราจะเขียน Search tree โดยให้แต่ละ node แสดง $t_0t_1...t_n$ โดย t_i แทน column ที่วาง Queen ในแถวที่ i จะได้ออกมาดังนี้



จาก tree หากเราดู Partial solution 01 จะเห็นว่า Queen นั้นอยู่ในตำแหน่งที่ไม่เหมาะสม เพราะทางเดินของ Queen ทั้งสองตัวซ้อนทับกันอยู่ในแนวเฉียง ดังนั้น การทำ Backtracking สำหรับ 4-Queen นี้ เราจะทำการตรวจสอบว่า Queen ตัวที่กำลังจะวางอยู่ในแนวเดินเฉียงของ Queen ที่อยู่ก่อนหน้าหรือไม่ โดยการพิจารณาว่า ถ้า row ต่างกัน n column ก็ห้ามต่างกัน n ซึ่งสามารถตัด tree ไปได้ประมาณครึ่งหนึ่งเลยทีเดียว



Ex3 : Sum of subset

กำหนด Input : Array D ซึ่งเก็บจำนวนเต็มบวก และจำนวนเต็ม K

Output : Subset E ของ D ที่ผลรวมของสมาชิกทุกตัวใน E มีค่าเท่ากับ K

Search space ของปัญหานี้ก็คือ subset ทั้งหมดที่เป็นไปได้ของ D ซึ่งเราสามารถสร้างได้โดยการดูสมาชิกแต่ละตัวใน D แล้วตัดสินใจว่าจะเลือกให้อยู่ใน subset หรือไม่เลือก ถ้าหากเลือกก็ให้เอาค่าที่เลือกไปบวกกับค่าของ Partial solution ปัจจุบัน จะได้ search space ขนาด 2^n

กรณีของปัญหาข้อนี้ เราสามารถทำ Backtracking เพื่อลดขนาดของ Search space ได้ง่ายๆ โดยการตรวจสอบก่อนที่จะเลือกสมาชิกตัวถัดไปจาก D ว่าถ้าเราเลือกสมาชิกตัวนั้นด้วยแล้ว จะทำให้ Partial solution เกินค่า K หรือไม่ ถ้าเกินก็ไม่ต้องทำกรณีนั้น

เราอาจสร้างตัวแปร total ไว้เก็บว่าทั้งหมดที่เลือกมาตอนนี้ ได้ผลรวมเท่าไหร่ จะได้ไม่ต้องมาหา sum ทุกครั้งที่เรียกด้วย

Branch & Bound

- เหมาะกับ optimizing problem
- maximization/minimization
- ใช้ bound เป็นตัวนำทางในการค้นด้วย

Idea จะเหมือนกับวิธี Backtracking แต่เราจะไม่สนใจเฉพาะ Partial solution เท่านั้น เราจะสนใจไปถึง Solution ที่จะโดน generate มาจาก Partial solution ด้วย วิธีนี้จึงเหมาะกับปัญหาที่เกี่ยวกับค่าที่ดีที่สุดซึ่งต้องดูทั้ง Search space

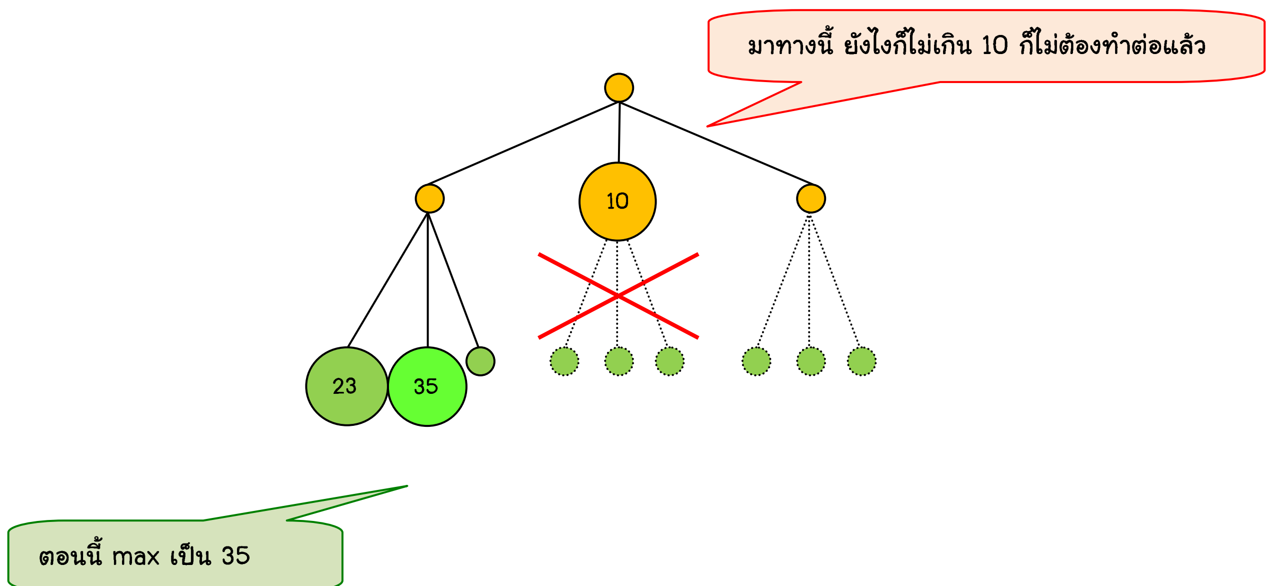
มาลองพิจารณา Maximizing Problem

เราจะต้องการ **Bounding Heuristic** ซึ่งก็คือ function ที่ตอบว่า ใน partial solution ที่เราทำอยู่ มีสิ่งที่ยังไม่ได้ทำที่ดีที่สุดเท่าไหร่ ถ้าผลรวมของ partial solution กับค่าของ function นี้ ยังน้อยกว่าค่า max ที่เคยหามาได้ ก็ไม่ต้องไปทางนี้ต่อแล้ว

จะเห็นว่า Branch & Bound เป็น special version ของ Backtracking โดย Branch & Bound จะสนใจอนาคตด้วย ในขณะที่ Backtracking จะดูแค่ว่าตอนนี้เป็นอย่างไร

Ex1 : **Maximum sum**

สมมติว่าเรากำลังหาค่า sum ของ sub sequence จาก sequence หนึ่งอยู่ และปัจจุบันมีค่า max อยู่จำนวนหนึ่ง เรากำลังจะ generate solution จาก Partial solution ปัจจุบัน ก็เช็คก่อนว่า ค่า sum ปัจจุบันเพื่อรวมกับค่าที่ได้จาก Bounding function เป็นอย่างไรเมื่อเทียบกับ max ถ้าพบว่าน้อยกว่า ก็ไม่ต้องไป generate solution ต่อไปอีกสำหรับ Partial solution นี้



Ex2 : Knapsack

ปัญหา คือ เลือกหยิบของใส่ถุงให้ได้ value สูงที่สุดโดยที่ weight รวมไม่เกินที่ถุงรับได้ สามารถแก้ได้หลายวิธี

วิธีธรรมดา :

เลือกว่าของแต่ละชิ้นนั้น จะเอา/ไม่เอา

ถ้าได้มูลค่ามากกว่า max และน้ำหนักไม่เกิน ก็เอา แล้วเปลี่ยน max

Backtracking :

ถ้าน้ำหนักเกิน ก็หยุด ไม่ไปทางนั้นต่อ (คล้ายๆ Sum of Subset ที่ถ้าผลรวมเกินก็หยุด)

Branch & Bound :

เราจะมีค่า max อยู่ และรู้ sum value ของตอนนี้ และรู้ด้วยว่า sum value ทั้งหมดที่ยังไม่พิจารณา คือเท่าไหร่

ถ้า sum value ของที่เหลือ + sum value ของตอนนี้ ยังน้อยกส่า max ที่เคยเจอ ก็ไม่ต้องทำต่อแล้ว

สมมติว่าปัจจุบันมูลค่าของสูงสุดเป็น 100 กำลัง search อยู่ใน step ที่ 5 และค่า sum ปัจจุบันเป็น 20 ก็ทดสอบดูว่า หากใน step ถัดๆไปเราเลือกของทุกชิ้นโดยไม่คำนึงถึงน้ำหนัก แล้วยังมูลค่าที่ได้ไม่ถึง 100 ก็ไม่ต้องไป generate solution แล้ว

~*------------------------*~