Complete search

- Complete search เป็นวิธีการทั่วไปที่สามารถนำไปใช้ในการแก้ปัญหาทางอัลกอริทึมได้เกือบหมด
- แนวคิดของมันคือการลองผลเฉลยของปัญหาทุกแบบที่เป็นไปได้โดยใช้การ brute force จากนั้นเลือกเอาผลเฉลยที่ดีที่สุดหรือนับจำนวนผลเฉลยที่เกิดขึ้นได้ ซึ่งขึ้นกับปัญหา
- Complete search เป็นเทคนิคที่ดี ถ้ามีเวลาเพียงพอในการไปลองทุกผลเฉลย เพราะว่าการค้นหาโดยทั่วไป implement ง่ายและให้คำตอบที่ถูกต้อง ถ้า complete search ทำงานช้าเกินไป ก็ต้องลองวิธีอื่นเช่น greedy dynamic

Generating subsets

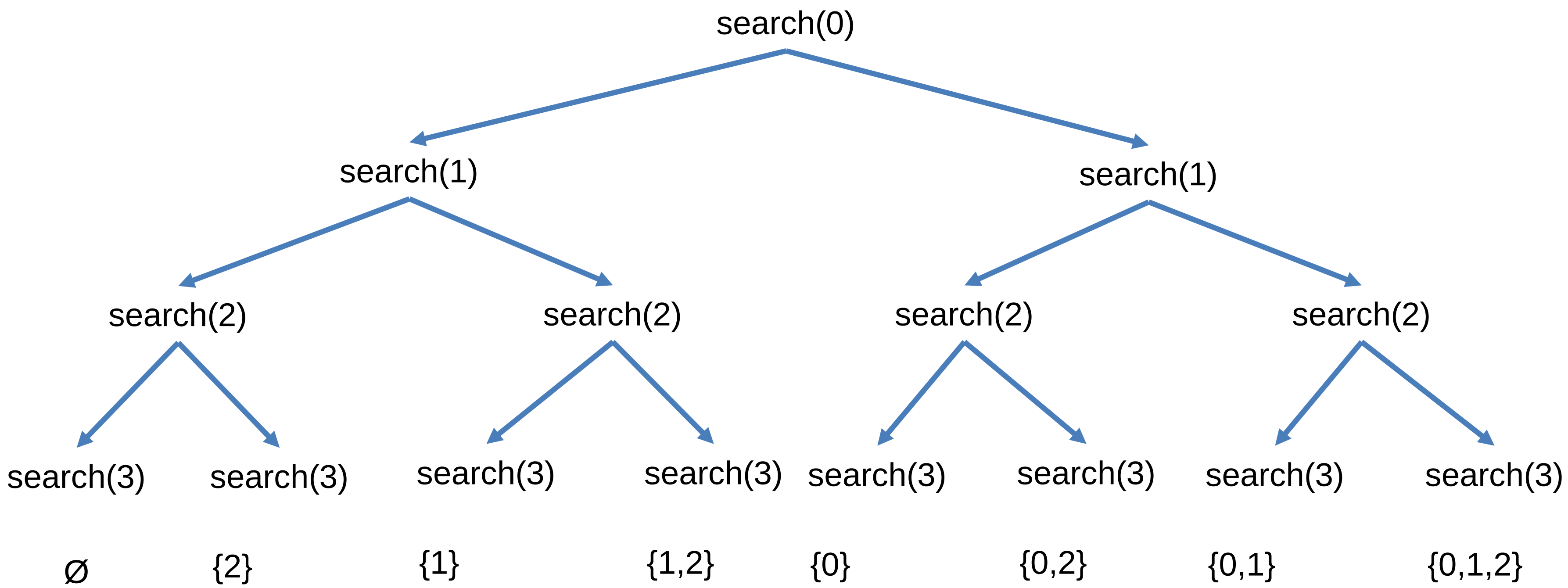
- เราจะเริ่มพิจารณาปัญหาการสร้าง subset ทั้งหมดของเซตที่มีสมาชิก n ตัว ตัวอย่างเช่น subset ของ {0,1,2} คือ {}, {0}, {1}, {2},{0,1},{0,2},{1,2},{1,2,3}
- มีวิธีที่คล้ายกันสองวิธีในการสร้าง subset เราสามารถสร้างโดยใช้ recursive search หรือการค้นหาโดยใช้ bit ของ integer

Method1

- วิธีแรก วิธีที่สวยในการลองทุก subset ของ set คือการใช้ recursion ฟังก์ชันต่อไปนี้ search จะสร้าง subset ของ set {0,1,...n−1}
- ฟังก์ชันเก็บ vector subset ที่เก็บสมาชิกของแต่ละ subset
- ฟังก์ชัน search เริ่มต้นเมื่อฟังก์ชันถูกเรียกด้วย parameter 0

```
void search(int k) {
    if (k == n) {
        // process subset
    } else {
        search(k+1);
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
    }
}
```

- เมื่อฟังก์ชัน search ถูกเรียกด้วย parameter k มันจะตัดสินใจว่าจะรวมสมาชิก k เข้าไปใน subset ด้วยหรือไม่ จากนั้นทั้งสองกรณีจะเรียกตัวมันเองด้วย parameter k+1
- เมื่อ k=n ฟังก์ชันสังเกตว่าทุกสมาชิกได้ถูกประมวลผลและ subset ถูกสร้าง
- ต่อไปเป็น tree แสดงการเรียกฟังก์ชันเมื่อ n=3 subtree ลูกทางซ้ายคือไม่รวม k ส่วน subtree ลูกทางขวาคือนับรวม k


search(0)
search(1)
search(1)
search(2)
search(2)
search(2)
search(2)
search(3)
search(3)
search(3)
search(3)
search(3)
search(3)
search(3)
search(3)
∅
{2}
{1}
{1,2}
{0}
{0,2}
{0,1}
{0,1,2}

Method2

- อีกทางหนึ่งในการสร้าง subset อยู่บนพื้นฐานของการแทนเลขจำนวนเต็มด้วย bit แต่ละ subset ของเซตที่มีสมาชิก n ตัว สามารถถูกแทนได้ด้วยลำดับของ n bits ซึ่งสอดคล้องกับเลขจำนวนเต็มระหว่าง $0 - 2^n-1$ โดยที่ 1 แทนว่าสมาชิกตัวนี้ถูกรวมใน subset
- ส่วนใหญ่แล้วจะให้ bit สุดท้ายแทนสมาชิก 0 บิตรองสุดท้ายแทนสมาชิก 1 ต่อไปเรื่อยๆ ตัวอย่างเช่น การแทน 25 ด้วย bit คือ 11001 ซึ่งจะสอดคล้องกับ subset {0,3,4}

- code ต่อไปเป็นการเข้าไปถึงทุก subset ของ set ที่มี n สมาชิก

```
for (int b = 0; b < (1<<n); b++) {
// process subset
}
```

1<<n คือ 1 แล้ว shift bit ไป n bit นั่นคือถ้า n=3

จาก 00000001 จะเปลี่ยนไปเป็น 00001000

ข้างต้นทำให้เรา print 0-7

- code ต่อเป็นเป็นตัวอย่างการหาว่ามีสมาชิกของ subset ที่สอดคล้องกับลำดับ bit นั่นคือเมื่อประมวลผลแต่ละ subset แล้วจะสร้าง vector ไว้เก็บสมาชิก

```
int n=3;
int s[3]={0,3,4};
for (int b = 0; b < (1<<n); b++) {
    vector<int> subset;
    for (int i = 0; i < n; i++) {
        if (b&(1<<i)) {
            subset.push_back(i);
        }
    }
    for(auto &l:subset){
        cout<<s[l]<<" ";
    }
    cout<<endl;
}
```

Generating permutations

- ต่อมาเราจะมาดูวิธีการสร้าง permutation ของเซตที่มีสมาชิก n ตัว ตัวอย่างเช่น permutation ของ {0,1,2} คือ (0,1,2),(0,2,1),(1,0,2),(1,2,0),(2,0,1) และ (2,1,0)
- ในที่นี้จะทำให้ดู 2 วิธีคือ recursion และ iterative

Method1

- คล้ายกับการสร้าง subset permutation นั้นถูกสร้างได้โดยการใช้ recursive ฟังก์ชันต่อไป search จะเป็นการสร้าง permutation ของเซต {0,1,...n−1} ฟังก์ชันนี้สร้าง vector permutation ที่เก็บ permutation
- search เริ่มต้นโดยเรียกไม่มีการส่ง parameter

```cpp
vector<int> permutation;
int n=3;
bool chosen[3];
void search() {
    if (permutation.size() == n) {
        for(auto &i:permutation)
            cout<<i<<" ";
        cout<<endl;
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permutation.push_back(i);
            search();
            chosen[i] = false;
            permutation.pop_back();
        }
    }
}
```

- การเรียกฟังก์ชันแต่ละครั้งจะเพิ่มสมาชิกใหม่ให้กับ permutation
- ส่วน array chosen นั้นเป็นตัวบอกว่าสมาชิกตัวใดที่ถูกรวมไปแล้วใน permutation บ้าง
- และเมื่อไรที่ขนาดของ permutation เท่ากับขนาดของเซต permutation ก็จะสมบูรณ์และถูก generate

Method2

- อีกวิธีหนึ่งในการสร้าง permutation คือเริ่มต้นด้วย permutation {0,1,2,...n−1} และจากนั้นทำซ้ำโดยใช้ฟังก์ชันที่สร้าง permutation ในลำดับที่มากขึ้น
- ข่าวดี ใน C++ มี standard library ที่เก็บ next permutation

```
vector<int> permutation;
    for (int i = 0; i < 3; i++) {
        permutation.push_back(i);
    }
    do {
        for(auto &i:permutation)
            cout<<i<<" ";
        cout<<endl;
    } while (next_permutation(permutation.begin(),permutation.end()));
```

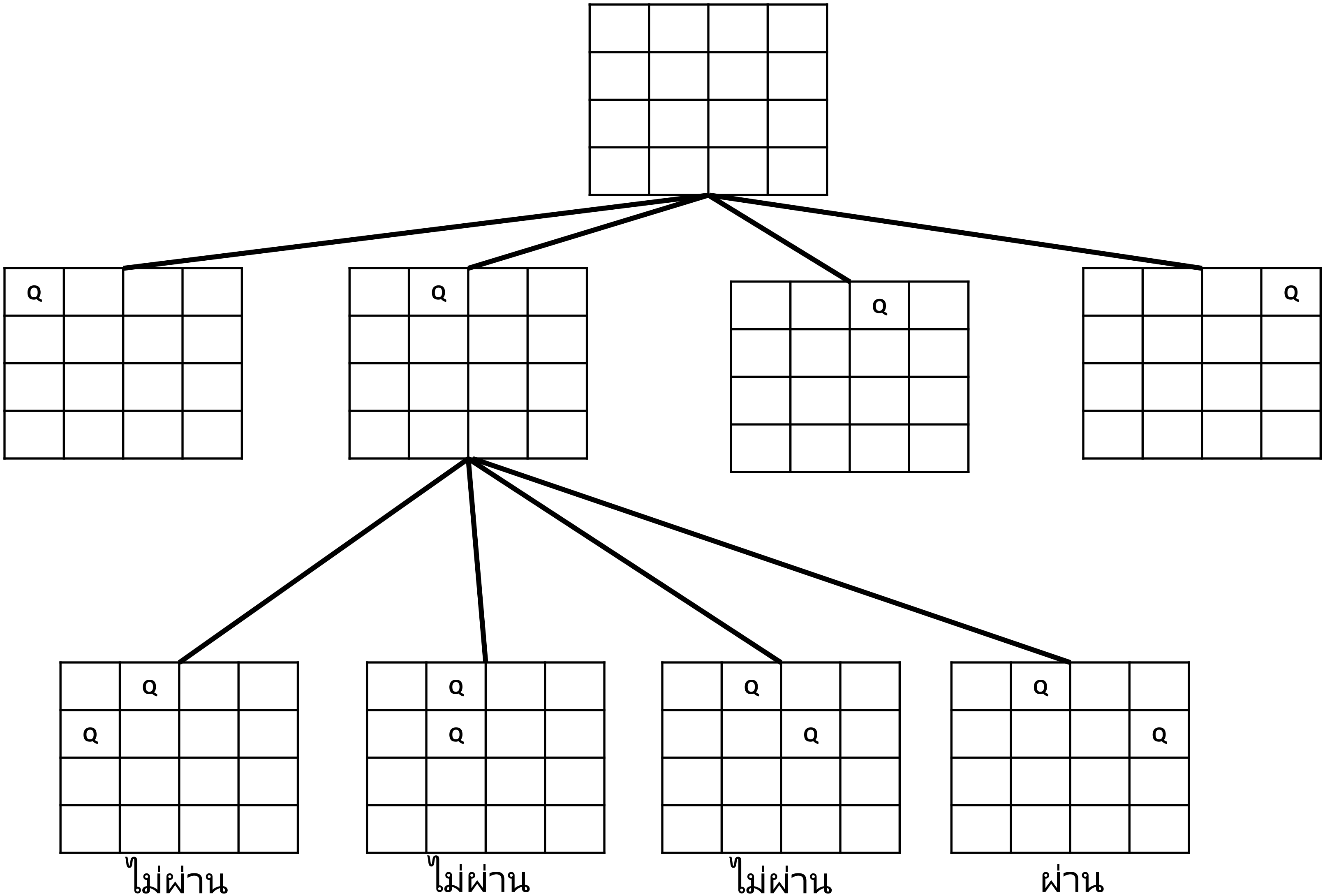
Backtracking

- Backtracking algorithm เริ่มต้นด้วย solution เปล่าและจะค่อยๆ ขยาย solution ทีละขั้น โดยการค้นหาจะ recursive ไปทุกเส้นทางที่แตกต่างกันเพื่อสร้างคำตอบ
- ตัวอย่างต่อไปเป็น การพิจารณาวิธีการคำนวณว่าจำนวนวิธีที่ queen n ตัวจะถูกวางได้ในตารางหมากรุกขนาด nxn โดยที่ไม่มี queen 2 ตัวใดๆ อยู่ในแนวกินกัน ตัวอย่างเช่น n=4 จะได้คำตอบ 2 แบบ

	Q		
			Q
Q			
		Q	

		Q	
Q			
			Q
	Q		

- ปัญหา n queen นี้สามารถแก้ได้โดย backtracking โดยการวาง queen ลงตารางทีละแถว พูดให้ชัดเจนคือ queen ตัวหนึ่งจะถูกวางลงแต่ละแถว โดยที่ว่าจะไม่มี queen ตัวอื่นที่วางก่อนหน้ากินได้
- คำตอบจะถูกพบเมื่อ queen n ตัวถูกวางบนตารางแล้ว
- ตัวอย่างเมื่อ n = 4 ตัวอย่างคำตอบที่ได้จากการใช้ backtracking algorithm


Q
Q
Q
Q
Q
Q
Q
Q
Q
Q
Q
Q
ไม่ผ่าน
ไม่ผ่าน
ไม่ผ่าน
ผ่าน

- ในชั้นล่างจะพบว่า รูปแบบสามอันแรกนั้นไม่ผ่าน เพราะว่า queen กินกันได้ อย่างไรก็ตามรูปแบบที่สี่นั้นผ่าน และมันสามารถขยายต่อไปจนได้คำตอบที่สมบูรณ์ได้ โดยการวาง queen อีกสองตัวในกระดาน ทั้งนี้มีอีกเพียงวิธีเดียวในการวาง queen สองตัวที่เหลือ

```
void solveNQUtil(int board[N][N], int col)
{   if (col >= N){
        //return true;
        c++;
    }
    for (int i = 0; i < N; i++)
    {
        if ( isSafe(board, i, col) )
        {
            board[i][col] = 1;
            solveNQUtil(board, col + 1);
            board[i][col] = 0; // BACKTRACK
        }
    }
}
```

```
int main()
{
    int board[N][N] = { {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0}
    };
    solveNQUtil(board, 0) ;
    cout<<c;
    return 0;
}
```

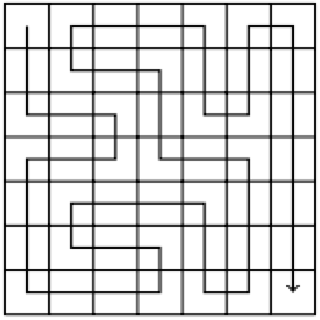
และมี global variable เป็น

```
#define N 4
int c=0;
```

- **ให้เขียน function issafe !!!!!!!!!!!!!!!!!!!**
- การค้นหาเริ่มต้นการการเรียก solveNQUtil(board, 0) ; หลังจากนั้น code จะทำการคำนวณว่ามีคำตอบกี่แบบเก็บไว้ที่ตัวแปร c
- code จะมี base case เป็นกรณีที่สามารถวาง queen ได้ครบ n ตัว ซึ่งเราจะนับแบบของคำตอบในกรณีนี้
- ส่วนกรณีอื่น เราฟิก col แล้วพยายามวาง queen ในทุกแถวทีละตัวถ้าวางได้ เมื่อวาง queen แล้วเราจะเรียก solveNQUtil(board, col+1) เพื่อวาง queen ใน col ต่อไปซึ่งก็จะไปลองกับทุกแถว ในการตรวจว่าวางได้หรือไม่ใช้ฟังก์ชัน issafe เมื่อวางเสร็จเราจะเอา queen ออกเพื่อลองแบบอื่น

Pruning the search

- เราสามารถ optimize backtracking โดยการ pruning search tree แนวคิดคือ เพิ่ม**ความฉลาด**ให้กับ algorithm เพื่อที่ว่ามันจะสังเกตเห็นเร็วที่สุดว่าคำตอบย่อยนั้นไม่สามารถนำไปสร้างเป็นคำตอบสุดท้ายได้ ซึ่งการ optimize นี้ทำให้เพิ่มประสิทธิภาพได้มาก
- พิจารณาปัญหาการคำนวณจำนวนเส้นทางที่เป็นไปได้ในตารางขนาด nxn จากมุมบนซ้ายไปยังมุมล่างขวา โดยที่เส้นทางนั้นผ่านแต่ละช่องเพียงครั้งเดียว ตัวอย่างเช่นตารางขนาด 7x7 มีรูปแบบ 111712 เส้นทาง ตัวอย่างหนึ่งคือ



- เราโฟกัสที่กรณี 7x7 เราจะเริ่มต้นด้วย backtracking algorithm แบบตรงไปตรงมาก่อน จากนั้นเราจะ optimize มันทีละขั้น โดยใช้การสังเกตว่าการค้นหาสามารถถูกตัดทิ้ง (prun) ได้

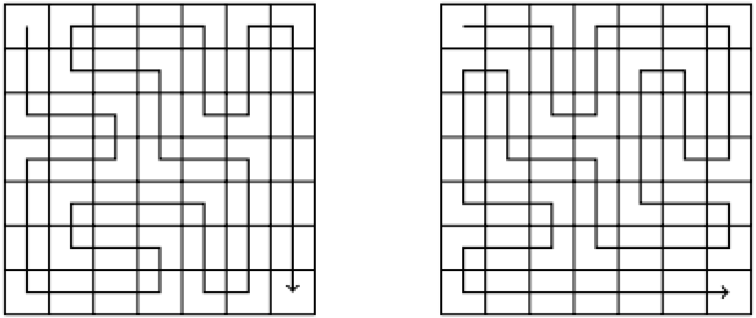
- หลังจากการ optimize แต่ละครั้งเราจะวัดเวลาในการทำงานของ algorithm และจำนวน recursive call เพื่อให้เห็นผลของการ optimize

Basic algorithm

- ใน version แรกของ algorithm ไม่มีการ optimize ใดๆ เราใช้เพียง backtracking ในการสร้างเส้นทางที่เป็นไปได้จากมุมบนซ้ายมามุมล่างขวา และนับจำนวนเส้นทาง พบว่าใช้เวลาไป 483 วินาที และมีจำนวนการเรียก recursive ทั้งหมด 76 พันล้านครั้ง

Optimization1

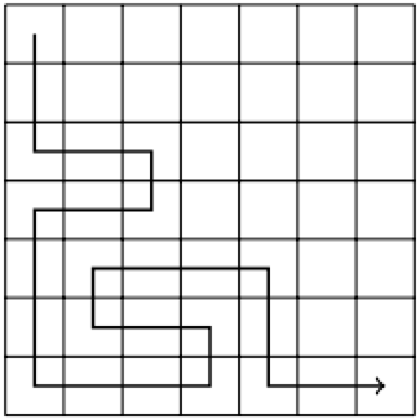
- ในคำตอบใดๆ เราพบว่าในช่องแรกเราจะเคลื่อนที่ไปทางขวาไม่ก็เคลื่อนที่ลง ซึ่งพบว่าจะมี 2 เส้นทางที่สมมาตรกันทางเส้นทแยงมุมของตารางเสมอหลังจากขั้นแรก ตัวอย่างเช่น



- ดังนั้นเราสามารถตัดสินได้ว่าเราจะเดินลงก่อน(หรือทางขวาก่อน) แล้วคูณจำนวนที่เป็นไปได้ทั้งหมดด้วยสอง พบว่าทำแบบนี้ใช้เวลา 244 วินาที และมีการเรียก recursive 38 พันล้านครั้ง

Optimization2

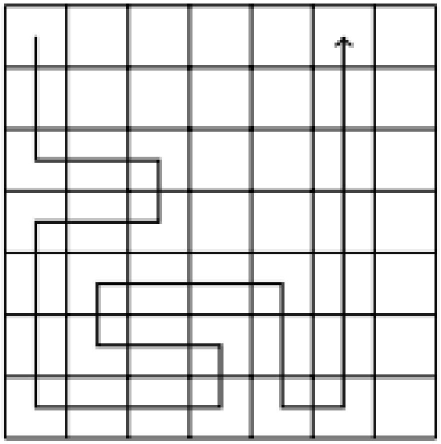
- ถ้าเส้นทางไปถึงมุมล่างขวาของสี่เหลี่ยมก่อนที่จะไปแวะช่องที่เหลือของตาราง นั่นแสดงว่ามันไม่สามารถเป็นคำตอบที่สมบูรณ์ได้ ตัวอย่างเช่น



- จากการสังเกตนี้ เราสามารถหยุดการค้นหาได้ทันทีถ้าเราไปถึงมุมล่างขวาเร็วเกินไป
- พบว่าเวลาในการทำงานเป็น 119 วินาที เรียก recursive 20 พันล้านครั้ง

Optimization3

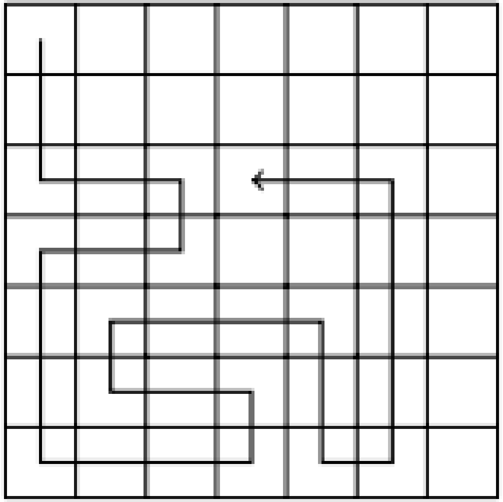
- ถ้าเส้นทางชนขอบและสามารถหันได้ทั้งซ้ายและขวา ตารางจะถูกแบ่งออกเป็นสองส่วนซึ่งเราสามารถไปได้ฝั่งเดียว ซึ่งในกรณีนี้เราไม่สามารถแวะทุกช่องได้ เราก็หยุดการค้นหาได้



- กรณีนี้ช่วยได้มาก เวลาในการทำงาน 1.8 วินาที จำนวน recursive 221 ล้านครั้ง

Optimization4

- จากแนวคิดของ optimization3 ถ้าเส้นทางไม่สามารถไปต่อได้แต่สามารถเลี้ยวได้ทั้งซ้ายและขวา ตารางจะถูกแบ่งออกเป็นสองส่วนที่มีส่วนหนึ่งไม่สามารถไปแวะได้



- หลังจาก optimize กรณีนี้ไป เวลาในการทำงานเหลือ 0.6 วินาทีและจำนวน recursive call เหลือ 69 ล้านครั้ง

- Running time เริ่มต้นเป็น 483 วินาที หลังจาก optimization แล้ว running time เหลือเพียง 0.6 วินาที ดังนั้น algorithm เร็วขึ้นเกือบ 1000 เท่าเลย
- นี้เป็นสิ่งที่เกิดขึ้นใน backtracking เพราะว่า search tree ส่วนใหญ่จะโต มาก และแม้ว่าการสังเกตง่ายๆ ก็ทำให้การ prune การค้นหาก็มีประสิทธิภาพมาก
- โดยเฉพาะ optimization ที่เกิดขึ้นในขั้นแรกของ algorithm จะช่วยให้ลดได้มาก เพราะว่าอยู่บนส่วนบนของ search tree

Meet the middle

- Meet the middle เป็นเทคนิคเมื่อ search space ถูกแบ่งออกเป็นสองส่วนที่มีขนาดเท่ากันได้ การแบ่งการค้นหาถูกทำทั้งสองส่วนและจากนั้นจะนำเอาคำตอบมารวมกัน
- เทคนิคนี้ถูกใช้ถ้ามีวิธีที่ดีในการรวมคำตอบของการค้นหา ซึ่งในกรณีนี้การค้นหาทั้งสองนั้นใช้เวลาน้อยกว่าการค้นหาก้อนใหญ่ทีเดียว โดยทั่วไปเราจะแบ่งส่วนของ 2^n เป็น $2^{n/2}$ โดยใช้เทคนิคนี้
- ตัวอย่างเช่น พิจารณาปัญหาที่เมื่อเราได้รับ list ของจำนวนที่ยาว n ตัว และจำนวน x จากนั้นเราต้องการหาว่าจะสามารถหาเลขบางตัวที่นำมารวมกันแล้วได้ x หรือไม่

- ตัวอย่างเช่น list [2,4,5,9] และ x=15 เราสามารถเลือก [2,4,9] เพื่อที่จะได้ 2+4+9=15 อย่างไรก็ตามถ้า x=10 เราพบว่าไม่สามารถทำได้
- algorithm แบบง่ายของปัญหานี้คือการสร้างทุก subset จากนั้นตรวจสอบถ้าผลรวมของ subset ใดๆ มีค่าเป็น x เวลาในการทำงานก็จะเป็น $O(2^n)$ เพราะว่ามี 2^n subset หากใช้เทคนิค meet the middle เราจะได้ algorithm ที่มีประสิทธิภาพดีกว่า $O(2^{n/2})$

- แนวคิดคือแบ่ง list ออกเป็นสองส่วน A และ B ที่เก็บจำนวนครึ่งหนึ่ง การหาครั้งแรก จะสร้างทุก subset ของ A และเก็บผลรวมไว้ที่ list SA ในการหาครั้งที่สองเราจะสร้าง list SB จาก B
- หลังจากนั้นเราก็ตรวจสอบได้ว่ามันสามรถเลือก element หนึ่งจาก SA และอีกตัวจาก SB ที่ผลรวมเป็น x
- ตัวอย่างเช่น list [2,4,5,9] และ x=15 เริ่มต้นเราจะแบ่ง list เป็น A=[2,4] และ B=[5,9] จากนั้นเราจะสร้าง list SA=[0,2,4,6] และ SB=[0,5,9,14]
- ในกรณีนี้ sum=15 สามารถหาได้ เพราะว่า SA มีผลรวมเป็น 6 และ SB มีผลรวมเป็น 9 สอดคล้องกับคำตอบ [2,4,9]

- เวลาในการทำงานเป็น $O(2^{n/2})$ เพราะว่าทั้ง list A และ list B มีจำนวน n/2 และใช้เวลา $O(2^{n/2})$ ในการคำนวณ sum ของ subset ของ SA และ SB จากนั้นสามารถตรวจสอบได้ใน $O(2^{n/2})$ ถ้าผลรวม x สามารถถูกสร้างได้จาก SA และ SB