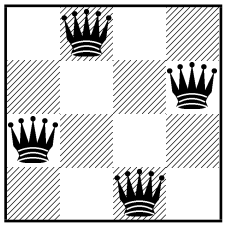
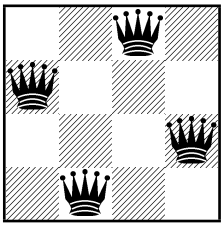
NQUEEN

1. 소개. 이 프로그램은 Jeff Somers[1]가 작성한 "N 개의 여왕 문제"의 해의 갯수를 구하는 프로그램을 CWEB로 재작성한 것이다.

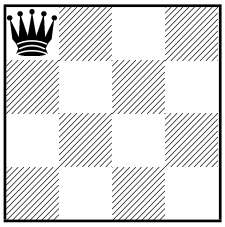
N 개의 여왕문제는 가로 세로의 크기가 N 인 체스판에 N 개의 여왕들을 배치하는 방법에 대한 문제인데, 여왕들을 배치할 때, 각 여왕들이 서로를 공격하지 못하도록 배치해야 한다. 체스의 여왕은 우리 장기의 차(車)와 그 기능이 비슷하고, 가로 세로 뿐만 아니라 대각선으로도 공격할 수 있는 점이 다르다. 예를 들어, 4×4 크기, 즉 N 이 4 인 경우의 체스판에 4 개의 여왕들을 배치하는 방법은 아래와 같이 두가지만이 존재한다. 직접 확인해 보기 바란다.



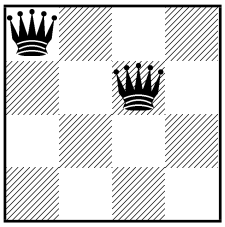


사실 N 개의 여왕문제 자체는 그리 흥미로운 문제는 아니지만 이 문제는 알고리즘 최적화에서 테스트베드로 사용되는 문제이다. 따라서 얼마나 빠른 시간에 이 문제를 해결하느냐가 관건이다. 제프의 프로그램은 제프 자신의 말에 의하면, 23×23 크기의 문제에 대한 세계 기록[2] 보다 2 배 빠르고, Dr. Dobb's Journal 에 소개된 Timothy Rolfe 박사의 그것보다는 10 배가 빠르다고 한다.[3]

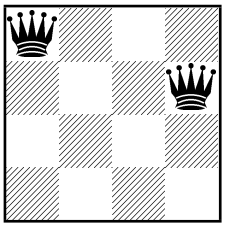
제프의 방법을 알아보기 앞서, 문제를 확실히 이해하고, 해결 방법을 알아보기 위해서, 직접 손으로 풀어보자. N 이 4 인 경우를 예로 든다. 순서대로 하기 위해서 여왕을 (1 행,1 열) 에 두고 시작한다.



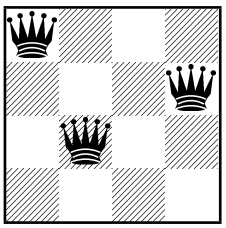
첫번째 여왕이 (1 행,1 열) 에 있기 때문에 두번째 여왕은 1 행 전체와 1 열 전체에는 올 수 없다. 또한 여왕은 대각선 공격도 가능하기 때문에 (2 행,2 열) 에도 올 수 없다. 따라서 두번째 여왕이 2 행에 올 수 있는 자리는 (2 행, 3 열) 과 (2 행, 4 열) 뿐이다. 이 두가지 경우 중에, 먼저 두번째 여왕을 (2 행, 3 열) 에 놓고 세번째 여왕이 올 수 있는 곳을 살펴보자.



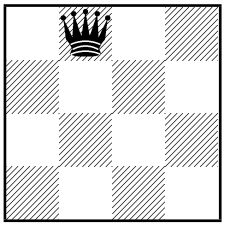
2. 세번째 여왕은 3 행에 오는데 1 열은 첫번째 여왕때문에 올 수 없고, 2 열, 3 열, 4 열은 두번째 여왕의 공격때문에 올 수 없다. 따라서 세번째 여왕이 놓일 수 있는 자리는 3 행에는 아무곳도 없다. 즉 뭔가가 잘못되었다. 아마도 두번째 여왕을 (2 행,3 열) 에 놓은 것이 잘못되었나보다. 다행히 두번째 여왕은 (2 행,4 열) 에도 올 수 있으므로, 두번째 여왕을 (2 해,4 열) 로 옮기고 다시 해보자. 이러한 방법을 퇴각기법 (Backtracking) 이라고 한다. 문제를 풀어나가다 막히면, 전 단계로 되돌아가서 전 단계의 다른 곳에 위치시켜서 문제를 해결하는 것이다. 이 퇴각기법은 미로에서 길을 찾는 데도 사용된다.



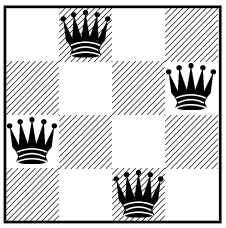
이제 세번째 여왕이 3 행에 올 수 있는 곳을 살펴보자. 먼저 1 열은 첫번째 여왕때문에 안되고, 3 열과 4 열은 두번째 여왕 때문에 안된다. 따라서 세번째 여왕이 올 수 있는 안전한 곳은 2 열밖에 없다.



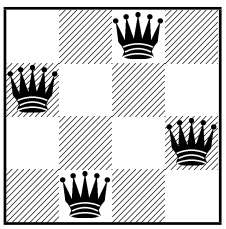
3. 마지막으로 네번째 여왕이 4 행에 놓일 수 있는 위치를 알아보자. 1 열은 첫번째 여왕때문에, 2 열과 3 열은 세번째 여왕때문에, 4 열은 두번째 여왕때문에 올 수 없다. 즉 네번째 여왕이 놓일 곳은 없다. 크기가 4 인 여왕문제는 해법이 없다는 말인가 ? 위에서 설명한 퇴각기법에 의거해 생각해보자. 현재 상황이 2 행과 3 행에 여왕이 놓일 수있는 위치는 그곳 밖에 없었다. 따라서 첫번째 여왕이 놓인 위치가 잘못된 것이다. 4 행을 처리하는 도중에 문제를 해결하기 위해서 퇴각 기법을 사용해보니, 1 행까지 갔다. 첫번째 여왕을 (1 행,2 열) 에 위치시키고 다시 해보자.



현재까지의 설명을 통해서 여왕을 안전하게 놓을 수 있는 방법을 알았으니, 이 경우에 해보면, 다음과 같은 해를 얻을 수 있다.



여러번의 시행 착오 끝에 위와 같은 한가지 해를 얻었다. 해가 더 있을지 모르니 해보자 첫번째 여왕을 3 열에 위치시키고 해보면, 아래와 같은 해를 얻을 수 있다.



마지막으로 (1 행,4 열) 에 위치시키고 해보면 (1 행,1 열) 에 했던 것처럼 해를 찾을 수 없다. 따라서 N 개의 여왕문제에서 N 이 4 인 경우는 위와 같은 두가지 해가 존재한다. 이제 감이 오는가 ? 퇴각 기법을 어떻게 구현하는가가 프로그램의 실행 속도를 좌우한다.

4. N 개의 여왕에 대한 문제의 해는 23×23 크기의 체스판에 대해서 까지 알려져 있다. 제프는 이 프로그램으로 21×21 크기까지 그 해를 구했는데, 여기에 걸린 시간은 800 MHz PC 에서 일주일 이상이 걸렸다고 한다. 이 해를 구하는 알고리즘의 시간 복잡도는 대략 $O(n!)$ 로 알려져 있고 22×22 크기의 체스판에 대한 해를 구하는데 걸리는 시간은 21×21 크기의 해를 구하는 것의

8.5 배 정도 걸린다. 심지어는 10GHz 의 컴퓨터로 계산한다고 하더라도, 23×23 의 해를 구하는 데는 한달 이상 걸릴 것이다. 물론 컴퓨터들을 클러스터로 묶고 계산하면 좀 더 걸리겠지만...

체스판 크기	해의 갯수	걸린 시간(800MHz PC)
1	1	N/A
2	0	< 0 초
3	0	< 0 초
4	2	< 0 초
5	10	< 0 초
6	4	< 0 초
7	40	< 0 초
8	92	< 0 초
9	352	< 0 초
10	724	< 0 초
11	2680	< 0 초
12	14200	< 0 초
13	73712	< 0 초
14	365596	00:00:01
15	2279184	00:00:04
16	4772512	00:00:23
17	95815104	00:02:38
18	666090624	00:19:26
19	4968057848	02:31:24
20	39029188884	20:35:06
21	314666222712	174:53:45
22	2691008701644	?
23	24233937684440	?
24	?	?

이제 컴퓨터로 해결하자. 역시 이러한 단순 반복 문제는 사람이 손으로 해결할 문제는 아닌것 같다. 제프의 퇴각기법 구현은 그 아이디어가 참 신선하다. 특히 비트연산을 이용한 방법과 스택을 이용한 방법은 배울만 하다.

5. 이 N 개의 여왕 배치 문제를 해결하는 프로그램의 전반적인 구조는 아래와 같다.

```
#include <stdio.h>
#include <stdlib.h>     /* atoi 함수 사용 */
#include <time.h>       /* 프로그램의 실행 시간을 측정 */
  ⟨ 체스판의 크기에 관한 매크로 및 전역 변수 6 ⟩
  ⟨ 함수들 8 ⟩
  int main(argc, argv)
       int argc;      /* 명령행의 인자 갯수 */
       char *argv[];      /* 명령행의 인자들 */
  {
    time_t t1, t2;      /* t1는 시작시점, t2는 종료시점의 시간을 측정한다 */
    int boardsize;      /* 체스판의 크기 */
    ⟨ 프로그램 사용법 24 ⟩;
    boardsize = atoi(argv[1]);
    ⟨ 체스판의 크기를 점검하라 7 ⟩;
    time(&t1);
    printf("N Queens program by Jeff Somers.\n");
    printf("\tallagash98@yahoo.com or jsomers@alumni.williams.edu\n");
    printf("Start:\t %s", ctime(&t1));
    Nqueen(boardsize);      /* 해를 구하는 부분으로 이 프로그램의 핵심이 되는 곳이다. */
    time(&t2);
    printResults(&t1, &t2);      /* 실행 시간 출력한다 */
    ⟨ 해의 갯수를 출력하라 23 ⟩;
    return 0;
  }
```

6. 체스판의 크기.

하나의 32 비트 `unsigned long` 크기의 변수는 체스판의 크기가 18×18 인 경우의 해의 갯수 (666090624 개) 를 담기에는 충분하지만, 19×19 인 경우 (4968057848 개) 를 담기에는 부족하다. Win32 의 경우에 결과를 담는 데에 제프는 64 비트 크기의 변수를 이용하여 이 프로그램에서 가질 수 있는 최대 체스판의 크기 `MAX_BOARDSIZE`를 21 로 하였다.

UNIX 의 경우에는 해의 갯수를 담는 전역 변수인 *g_numsolutions* 의 타입을 **unsigned long** 에서 **unsigned long long** 으로 바꾸거나, 혹은 이 변수를 32 비트 변수로 이용하면 19×19 크기까지의 해를 구할 수 있다.

⟨ 체스판의 크기에 관한 매크로 및 전역 변수 6 ⟩ ≡

```
#ifdef WIN32
#define MAX_BOARDSIZE 21
  typedef unsigned __int64SOLUTIONTYPE;
#else
#define MAX_BOARDSIZE 18
  typedef unsigned long SOLUTIONTYPE;
#endif
#define MIN_BOARDSIZE 2
  SOLUTIONTYPE g_numsolutions = 0;
```

이 코드는 5 번 마디에서 사용된다.

7.

체스판의 크기가 올바른지 확인한다. 그 크기는 2 보다는 커야하고 21 을 넘을 수는 없다. 21 보다 커도 되는데, 결과를 기다리려면 한 없이 기다려야 한다.

⟨ 체스판의 크기를 점검하라 7 ⟩ ≡

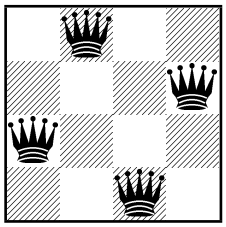
```
  if (MIN_BOARDSIZE > boardsize ∨ MAX_BOARDSIZE < boardsize) {
    printf("Width of board must be between %d and %d, inclusive.\n",
        MIN_BOARDSIZE, MAX_BOARDSIZE);
    return 0;
  }
```

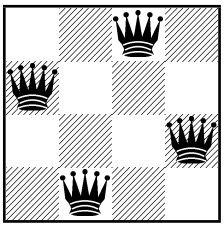
이 코드는 5 번 마디에서 사용된다.

---

atoi: **int** ( ), `<stdlib.h>`.
ctime: **char** ∗( ), `<time.h>`.
Nqueen: **void** ( ), §8.
printf: **int** ( ), `<stdio.h>`.
printResults: **void** ( ), §22.
time: **time_t** ( ), `<time.h>`.

8. N 개의 여왕 함수. 함수 $Nqueen(\,)$은 이 프로그램에서 핵심적인 부분으로 해를 구하는 함수이다. 프로그램 실행 속도 향상을 위해서 주어진 N 에 대해서 해를 구하는 것이 아니라 일단 N 의 반$(\lfloor N/2 \rfloor)$에 대해서 계산한 다음에 그 구한 해를 Y 축에 대해서 대칭으로 한다. 각각의 해는 Y 축에 대하여 대칭인 다른 고유의 해를 가지고 있으므로 이렇게 함으로 해서 모든 해를 구할 수 있는 것이다.(체스판의 크기가 1×1 인 경우는 제외) 무슨 말일고 하니, 그림으로 확인하자.





첫번째 해에서 2 열과 3 열 사이를 Y 축이라고 하고 대칭 시키면, 두번째 해를 얻을 수 있다. 즉 두번째 해는 첫번째 해에서 자동적으로 얻을 수 있는 것이므로 일부러 계산할 필요가 없는 것이다.

이처럼 Y 축에 대해서 대칭으로 나머지 반의 해를 구할 수 있는 이유는 (위의 예에서 두번째 해를 구할 수 있는 이유는) 모든 해는 스스로 Y 축에 대해서 대칭 일 수 없기 때문이다. 왜냐하면, 같은 행에 두개의 퀸을 놓을 수 없기 때문이다. 또한 해는 체스판의 크기가 홀수 일때는, 맨 가운데 열에 대해서만 별도로 따로 구하고, 다른 열들에 대해서는 1 열부터 (맨 가운데 열 -1) 까지의 해를 구해서 Y 축 대칭 시키면 홀수 일때도 모든 해를 구할 수 있다. 이것이 가능한 이유 역시 한 열에 두개 이상의 퀸이 올 수 없기 때문이다.

이 문제를 해결하는 알고리즘은 서두에서 설명했듯이 퇴각기법 (backtracking) 이다. 다시 한 번 더 설명하면, 먼저 (1 행,1 열) 에 여왕을 놓은 다음에 그 여왕의 세력이 미칠 수 있는 열과 대각선을 표시한다. 그리고나서 앞에서 표시된 열과 대각선을 피해서 다른 여왕을 배치시킨다. 이렇게 새로운 여왕이 놓임으로해서 이 두 여왕의 세력이 미치는 열과 대각선들을 새로 구한다. 이러한 방법으로 진행해 나가다가 만약 다음 행에 새로운 여왕을 놓을 위치가 없다면, 바로 전의 행으로 물러가서 여왕을 배치할 수 있는 다른 위치로 옮기고 계속 진행한다.

```
⟨ 함수들 8 ⟩ ≡
  void Nqueen(board_size)
       int board_size;
  {
    ⟨ Nqueen 함수의 지역 변수들 9 ⟩;
    ⟨ 스택을 초기화하라 11 ⟩;
    for (i = 0; i < (1 + odd); ++i) {
      bitfield = 0;
      if (0 ≡ i) {
        ⟨ 맨 가운데 열을 제외한 체스판의 오른쪽 반을 처리하기 위해 초기화하라 13 ⟩
      } else {
        ⟨ 체스판의 크기가 홀수 일때, 맨 가운데 열을 처리하기 위해 초기화하라 16 ⟩
      }
      ⟨ 퇴각 기법 알고리즘으로 문제를 해결하라 17 ⟩
    }      /* 현재까지의 결과를 두배하여 Y 대칭의 결과를 구한다. */
    g_numsolutions *= 2;
```

```
}
```

22번 마디도 살펴보라.

이 코드는 5번 마디에서 사용된다.

9. 이 프로그램에서는 실행의 최적화를 위해서 주로 비트 연산자를 이용한다. 예를 들며, 체스판의 크기가 8이고, 어떤 한 행에서 5번째 위치에 여왕을 놓을 수 있다면 그 행은 0000100으로 표현하고 따라서 그 행의 값은 이진수 0000100의 십진수 값인 4가 된다. 이러한 값을 갖는 변수가 바로 *bitfield*이다. 좀 더 자세히 설명하면, 변수 *bitfield*는 어느 순간 그 행에 놓을 수 있는 여왕의 위치를 표시하기 위해서 사용된다. 예를 들어 *bitfield*의 값이 15라면, *bitfield*는 15의 이진수인 00001111로 표시되고 따라서 그 행에 $5 \sim 8$ 열의 위치에 여왕을 놓을 수 있는 것이다.

또한 주목해야할 변수는 바로 *lsb* 인데, 이 변수가 나타내는 것은 한 행에서 여왕을 놓을 수 있는 맨 오른쪽 위치를 나타낸다. 이 프로그램은 오른쪽 반에 대해서 해를 구한 후에 그것을 Y 축에 대칭으로 하여 나머지를 구한다. 따라서 프로그램이 실행되는 순서는 맨 오른쪽부터 시작하여 왼쪽으로 옮겨간다. 따라서 변수 *lsb*는 현재 실행되고 있는 행의 몇번째 열에 여왕이 위치해 있는지를 나타낸다. 그리고 현재 실행 되고 있는 행을 나타내기 위해서 *numrows* 변수가 사용된다.

```
⟨ Nqueen함수의 지역 변수들 9 ⟩ ≡
  register int numrows = 0;      /* stack을 이용할 때 사용된다. */
  register unsigned int lsb;     /* 한 행의 맨 오른쪽 여왕의 위치 least significant bit */
  register unsigned int bitfield;     /* 여왕을 위치할 수 있는 비트 마스크 */
```

10, 12번 마디도 살펴보라.

이 코드는 8번 마디에서 사용된다.

10. 위의 설명에서 한 행의 여왕의 위치는 *bitfield*를 이용해서 나타낼 수 있다고 했다. 따라서 그 위치와 각 행에서 여왕의 세력이 체스판 전체에 미치는 영향을 표시해야 하는데 이때, 각종 배열이 사용된다. 각 배열의 하나의 원소가 현재 행의 여왕의 위치 및 세력 범위를 나타내므로 그러한 변수들로 이루어진 배열은 판 전체의 상황을 표시한다.

```
⟨ Nqueen함수의 지역 변수들 9 ⟩ +≡
  int aQueenBitRes[MAX_BOARDSIZE];     /* 결과를 저장 */
  int aQueenBitCol[MAX_BOARDSIZE];     /* 여왕이 세력을 미치는 열들을 표 */
  int aQueenBitPosDiag[MAX_BOARDSIZE];     /* 여왕이 세력을 미치는 양의 대각선을 표 */
  int aQueenBitNegDiag[MAX_BOARDSIZE];     /* 여왕이 세력을 미치는 음의 대각선을 표 */
  int aStack[MAX_BOARDSIZE + 2];     /* 스택을 사용한다. */
  register int *pnStack;     /* 스택 포인터 */
```

11. 이 프로그램에서는 퇴각기법을 구현하기 위해서 재귀적 방법을 사용하지 않고, 실행 속도 향상을 위해서 내부적으로 스택을 사용하여 구현한다.

```
⟨ 스택을 초기화하라 11 ⟩ ≡
  aStack[0] = -1;     /* 스택의 마지막을 표시 */
```

이 코드는 8번 마디에서 사용된다.

---

g_numsolutions: **SOLUTIONTYPE**, §6.

i: **int**, §12.

odd: **int**, §12.

12. 앞서 설명했듯이, 체스판의 크기가 홀수 일때는, 짝수 일때 처럼 크기의 반에 대해서만 구하고, 그것을 대칭하고, 맨 가운데 열을 처리하기 위해서 시작하여 한번 더 계산한다. 다만, 홀수 일때는 1 행의 위치의 여왕은 이미 맨 가운데 열로 결정되었으므로 두번째 행부터 계산해 나가면된다. 따라서 체스판의 크기가 짝수 일때는 퇴각 기법을 한번만 실행하면 되고, 홀수 일때는 두번을 실행하면 된다. 변수 *odd*의 값은 체스판이 홀수 이면 1, 짝수 이면 0이 되고, 전체 루프는 $(1 + odd)$ 만큼 반복된다.

⟨ *Nqueen*함수의 지역 변수들 9 ⟩ +≡
```
int i;
int odd = board_size & 1;      /* board_size가 짝수이면 0, 홀수 이면 1 */
int board_minus = board_size − 1;      /* board size - 1 */
int mask = (1 ≪ board_size) − 1;      /* 모두 1로 초기화 */
```

13. 가운데 열을 제외한 오른쪽 반을 처리한다. 만약 체스판의 크기가 5×5 라면 첫번째 행은 `00011`이 될것이다. 맨 가운데 열은 두번째 루프에서 처리할 것이므로 현재는 신경 쓰지 않아도 된다.

⟨ 맨 가운데 열을 제외한 체스판의 오른쪽 반을 처리하기 위해 초기화하라 13 ⟩ ≡
```
⟨ 첫행의 오른쪽 반을 모두 1로 마크하라 14 ⟩;
pnStack = aStack + 1;      /* 스택 포인터 */
⟨ 여왕의 세력 범위를 초기화하라 15 ⟩;
```
이 코드는 8번 마디에서 사용된다.

14. 여왕을 놓을 수 있는 위치를 나타내는 변수 *bitfield*의 오른쪽 반에 해당하는 비트들을 여왕이 있다는 표시인 1로 초기화 한다. 먼저 *board_size*를 반으로 나누고 $(half)$ 1을 비트 연산자를 이용하여 *half* 만큼 왼쪽으로 쉬프트하고 1을 빼면 오른쪽 반에 해당하는 비트를 모두 1로 만들 수 있다. 예를 들어 *board_size*가 7이면 *half*는 3이 되고, 1을 3만큼 왼쪽으로 시프트하면 `1000`이 되고, 여기서 1을 빼므로, *bitfield*의 값은 `111`이 된다. 판의 크기가 7이므로 `111`을 `0000111`로 간주하면 이해하기 쉽다.

⟨ 첫행의 오른쪽 반을 모두 1로 마크하라 14 ⟩ ≡
```
int half = board_size ≫ 1;      /* 비트 연산자를 이용해 2로 나눈다 */
bitfield = (1 ≪ half) − 1;
```
이 코드는 13번 마디에서 사용된다.

15. 초기에 여왕이 미치는 세력은 아무것도 없으므로 모두 0으로 초기화 한다.

⟨ 여왕의 세력 범위를 초기화하라 15 ⟩ ≡
```
aQueenBitRes[0] = 0;
aQueenBitCol[0] = aQueenBitPosDiag[0] = aQueenBitNegDiag[0] = 0;
```
이 코드는 13번 마디에서 사용된다.

16. 체스판의 크기가 홀수 일때는 1행의 맨 가운데 열을 나타내는 비트를 1로 세팅하고, 2행부터는 체스판이 $(N - 1) \times N$ 이라고 생각하고 기존 방법과 마찬가지로 오른쪽 반만 구해서 Y 축 대칭을 구한다. N 이 5일때를 예로 들면, 1행의 비트들은 `00100`이 되고, 2행은 `00011`이 된다.

⟨ 체스판의 크기가 홀수 일때, 맨 가운데 열을 처리하기 위해 초기화하라 16 ⟩ ≡
```
bitfield = 1 ≪ (board_size ≫ 1);
numrows = 1;      /* 1행의 맨가운데 열에는 이미 여왕이 있다 */
```

```
aQueenBitRes[0] = bitfield;
aQueenBitCol[0] = aQueenBitPosDiag[0] = aQueenBitNegDiag[0] = 0;
aQueenBitCol[1] = bitfield;
  /* 2 햄부터 처리한다: 2 행부터는 체스판이 (N − 1) × N 이라고 생각하고 처리한다 */
aQueenBitNegDiag[1] = (bitfield ≫ 1);
aQueenBitPosDiag[1] = (bitfield ≪ 1);
pnStack = aStack + 1;      /* 스택 포인터 */
*pnStack++ = 0;
bitfield = (bitfield − 1) ≫ 1;
```

이 코드는 8 번 마디에서 사용된다.

---

aQueenBitCol: **int** [], §10.
aQueenBitNegDiag: **int** [], §10.
aQueenBitPosDiag: **int** [], §10.
aQueenBitRes: **int** [], §10.
aStack: **int** [], §10.
bitfield: **register unsigned int**, §9.
board_size: **int**, §8.
Nqueen: **void** ( ), §8.
numrows: **register int**, §9.
pnStack: **register int** *, §10.

17. 퇴각기법 알고리즘. 이 프로그램에서 가장 중요한 부분이고, 연산량도 가장 많으므로 실행 속도 향상을 위해서 가장 최적화 되어야 할 부분이다. 알고리즘의 주요 흐름은 다음과 같다.

```
⟨ 퇴각 기법 알고리즘으로 문제를 해결하라 17 ⟩ ≡
  for ( ; ; ) {
    ⟨ 여왕이 놓일 수 있는 맨 오르쪽의 위치를 구하라 18 ⟩;
    if (0 ≡ bitfield) {      /* 더 이상 여왕을 놓을 위치가 없으면 */
      bitfield = *--pnStack;      /* 스택에서 바로 전행의 bitfield값을 가져온다 */
      if (pnStack ≡ aStack) {      /* 더이상 스택에서 꺼낼 값이 없다면 */
        break;
      }
      --numrows;
      continue;
    }
    ⟨ 현재 여왕의 위치를 0으로 하라 19 ⟩;
    aQueenBitRes[numrows] = lsb;      /* 현재 행에 대한 결과를 저장 */
    if (numrows < board_minus) {      /* 처리해야할 행들이 남아 있다면, */
      int n = numrows++;
      ⟨ 여왕의 세력 범위를 갱신하라 20 ⟩;
      ⟨ bitfield를 스택에 저장하고, 현재 행에서 여왕이 올수 있는 위치를 새로 구하라 21 ⟩;
      continue;
    } else {      /* 더이상 처리해야 할 행이 없으므로, 일단 정리한다. */
      ++g_numsolutions;      /* 해의 갯수를 1증가 */
      bitfield = *--pnStack;      /* 바로 전행의 다른 여왕의 위치에서 시작한다. */
      --numrows;
      continue;
    }
  }
```

이 코드는 8번 마디에서 사용된다.

18. 여왕이 놓일 수 있는 맨 오르쪽의 위치를 구하기 위해서 2의 보수 체계를 이용한다. 예를 들어서 어떤 순간의 *bitfield*가 00100100이라고 하자 이때의 *lsb*는 100이 된다. *lsb*는 맨 오르쪽에 있는 1비트이다 따라서 여왕이 올 수 있는 맨 오르쪽 위치이기도 하다. 00100100에서 100을 구하기 위해서 제프가 제시한 방법은 다음과 같다. 먼저 2의 보수 체계하에서 주어진 00100100를 음수로 한다. 2의 보수에서 음수는 주어진 비트들을 0은 1로 1은 0으로 바꾸고 1을 더하는 것이다. 따라서 00100100의 음수는 11011100이 된다. 이 둘을 비트 AND 연산을 취하면 100이 된다!

$$0010010 \;\&\; 11011100 = 100$$

```
⟨ 여왕이 놓일 수 있는 맨 오르쪽의 위치를 구하라 18 ⟩ ≡
  lsb = -((signed) bitfield) & bitfield;
```

이 코드는 17번 마디에서 사용된다.

19. *lsb* 즉 여왕이 올 수 있는 맨 오르쪽 위치를 구했으므로, 그 후 작업을 하면 되는데, 여기서는 *lsb*의 위치를 0으로 해서 다시 이 위치에서 여왕을 구하는 일이 없도록 한다.

```
⟨ 현재 여왕의 위치를 0으로 하라 19 ⟩ ≡
  bitfield &= ~lsb;
```

이 코드는 17번 마디에서 사용된다.

20. ⟨ 여왕의 세력 범위를 갱신하라 20 ⟩ ≡
```
aQueenBitCol[numrows] = aQueenBitCol[n] | lsb;
aQueenBitNegDiag[numrows] = (aQueenBitNegDiag[n] | lsb) ≫ 1;
aQueenBitPosDiag[numrows] = (aQueenBitPosDiag[n] | lsb) ≪ 1;
```
이 코드는 17 번 마디에서 사용된다.

21. ⟨ *bitfield*를 스택에 저장하고, 현재 행에서 여왕이 올수 있는 위치를 새로 구하라 21 ⟩ ≡
```
*pnStack++ = bitfield;
bitfield = mask & ~(aQueenBitCol[numrows] | aQueenBitNegDiag[numrows] |
    aQueenBitPosDiag[numrows]);
```
이 코드는 17 번 마디에서 사용된다.

---

aQueenBitCol: **int** [], §10.
aQueenBitNegDiag: **int** [], §10.
aQueenBitPosDiag: **int** [], §10.
aQueenBitRes: **int** [], §10.
aStack: **int** [], §10.
bitfield: **register unsigned int**, §9.
board_minus: **int**, §12.
g_numsolutions: **SOLUTIONTYPE**, §6.
lsb: **register unsigned int**, §9.
mask: **int**, §12.
numrows: **register int**, §9.
pnStack: **register int** ∗, §10.

22. 결과 출력. 실행 시간과 해의 갯수를 출력한다. 실행 시간은 *printResults* 함수를 이용한다.

```
⟨ 함수들 8 ⟩ +≡        /* Print the results at the end of the run */
  void printResults(time_t *pt1, time_t *pt2)
  {
    double secs;
    int hours, mins, intsecs;
    printf("End: \t%s", ctime(pt2));
    secs = difftime(*pt2, *pt1);
    intsecs = (int) secs;
    printf("Calculations took %d second%s.\n", intsecs, (intsecs ≡ 1 ? "" : "s"));
      /* Print hours, minutes, seconds */
    hours = intsecs/3600;
    intsecs -= hours * 3600;
    mins = intsecs/60;
    intsecs -= mins * 60;
    if (hours > 0 ∨ mins > 0) {
      printf("Equals ");
      if (hours > 0) {
        printf("%d hour%s, ", hours, (hours ≡ 1) ? "" : "s");
      }
      if (mins > 0) {
        printf("%d minute%s and ", mins, (mins ≡ 1) ? "" : "s");
      }
      printf("%d second%s.\n", intsecs, (intsecs ≡ 1 ? "" : "s"));
    }
  }
```

23. ⟨ 해의 갯수를 출력하라 23 ⟩ ≡

```
  if (g_numsolutions ≠ 0) {
#ifdef WIN32
    printf("For board size %d, %I64d solution%s found.\n", boardsize,
        g_numsolutions, (g_numsolutions ≡ 1 ? "" : "s"));
#else
    printf("For board size %d, %d solution%s found.\n", boardsize, g_numsolutions,
        (g_numsolutions ≡ 1 ? "" : "s"));
#endif
  }
  else {
    printf("No solutions found.\n");
  }
```

이 코드는 5 번 마디에서 사용된다.

24. 프로그램의 실행. 프로그램의 실행은 명령행 인자로 체스판의 크기만 주어지면 된다. N이 10인 경우는 다음과 같이 실행된다.

```
myhome% nq 10
N Queens program by Jeff Somers.
allagash98ahoo.com or jsomerslumni.williams.edu
Start:   Fri Apr 28 14:12:21 2006
End:  Fri Apr 28 14:12:21 2006
Calculations took 0 seconds.
For board size 10, 724 solutions found.
```

⟨ 프로그램 사용법 24 ⟩ ≡

```
if (argc ≠ 2) {
  printf("N␣Queens␣program␣by␣Jeff␣Somers.\n");
  printf("\tallagash98@yahoo.com␣or␣jsomers@alumni.williams.edu\n");
  printf("This␣program␣calculates␣the␣total␣number␣of␣solutions␣to␣th\
      e␣N␣Queens␣problem.\n");
  printf("Usage:␣nq␣<width␣of␣board>\n");      /* user must pass in size of board */
  return 0;
}
```

이 코드는 5번 마디에서 사용된다.

---

argc: **int**, §5.
boardsize: **int**, §5.
ctime: **char** *(), <time.h>.
difftime: **double** (), <time.h>.
g_numsolutions:
SOLUTIONTYPE, §6.
printf: **int** (), <stdio.h>.