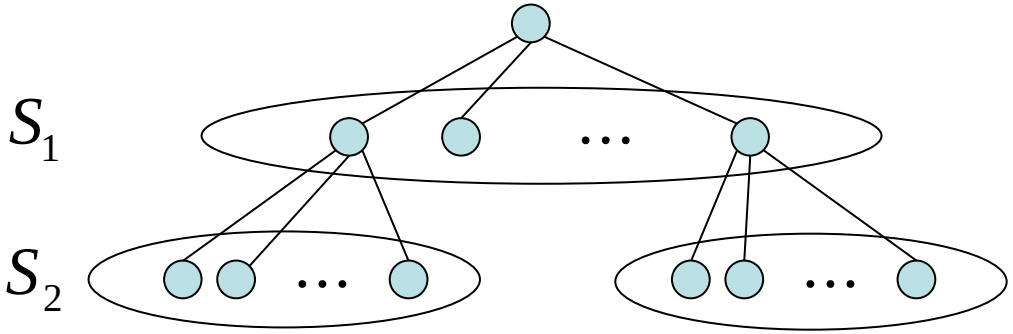
Backtracking

- Backtracking is a systematic method to iterate through all the possible configurations of a search space.
- It is a general algorithm/technique which must be customized for each individual application and is generally considered with a search tree.

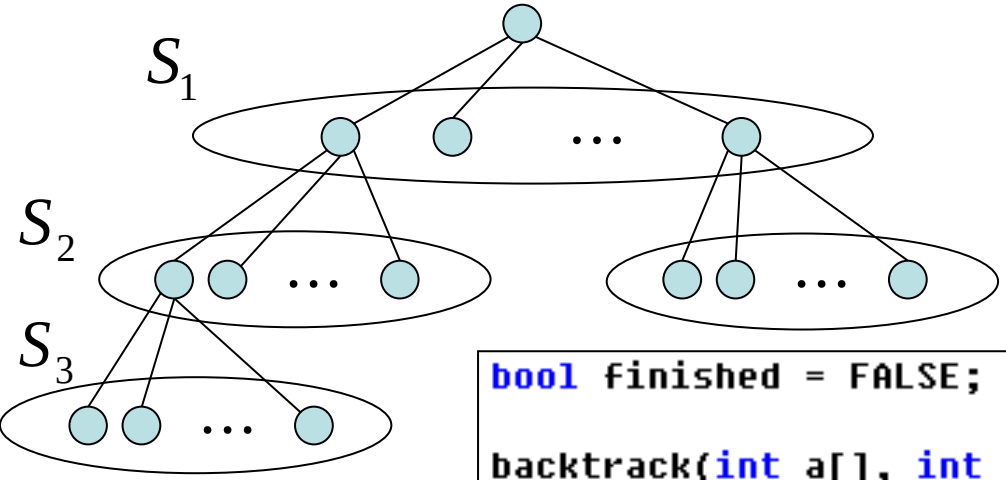


- In the general case, we will model our solution as a vector $a=(a_1, a_2,\ldots, a_n)$ where each element a_i is selected from a finite ordered set S_i.
- Such a vector might represent an arrangement where a_i contains the ith element of the permutation. Or the vector might represent a given subset S, where a_i is true if and only if the ith element of the universe is in S.

Implementation

- The honest working code is given below. We include a global `finished` flag to allow for premature termination, which could be set in any application-specific routine.

```
bool finished = FALSE;          /* found all solutions yet? */

backtrack(int a[], int k, data input)
{
   int c[MAXCANDIDATES];  /* candidates for next position */
   int ncandidates;       /* next position candidate count */
   int i;                 /* counter */

   if (is_a_solution(a,k,input))
      process_solution(a,k,input);
   else {
      k = k+1;
      construct_candidates(a,k,input,c,&ncandidates);
      for (i=0; i<ncandidates; i++) {
         a[k] = c[i];
         backtrack(a,k,input);
         if (finished) return;  /* terminate early */
      }
   }
}
```



Implementation

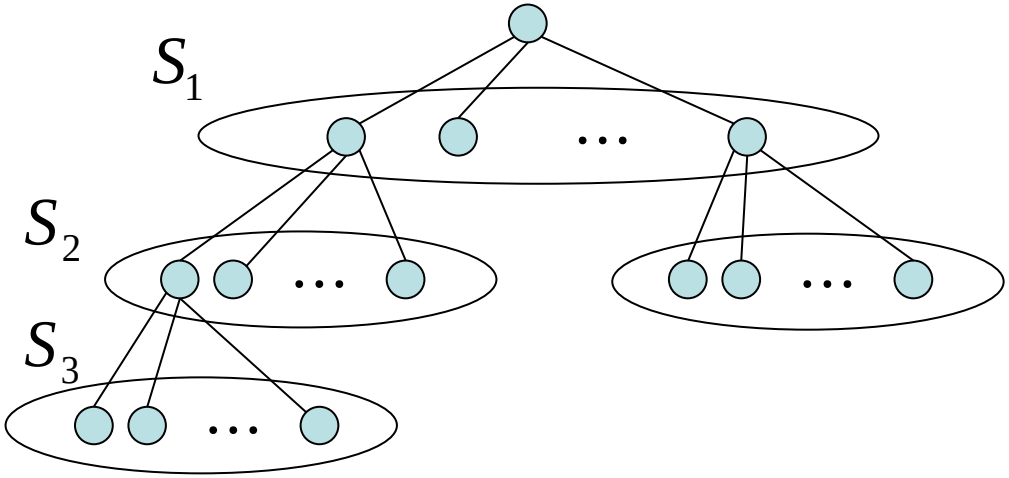
```
bool finished = FALSE;          /* found all solutions yet? */

backtrack(int a[], int k, data input)
{
   int c[MAXCANDIDATES];  /* candidates for next position */
   int ncandidates;       /* next position candidate count */
   int i;                 /* counter */

   if (is_a_solution(a,k,input))
      process_solution(a,k,input);
   else {
      k = k+1;
      construct_candidates(a,k,input,c,&ncandidates);
      for (i=0; i<ncandidates; i++) {
         a[k] = c[i];
         backtrack(a,k,input);
         if (finished) return;  /* terminate early */
      }
   }
}
```

Application-Specific Routines

- The application-specific parts of this algorithm consists of three subroutines:
- `is_a_solution(a,k,input)`-This Boolean function tests whether the first *k* elements of vector *a* are a complete solution for the given problem. The last argument, input, allows us to pass general information into the routine.
- `construct_candidates(a,k,input,c,ncandidates)`- This routine fills an array *c* with the complete set of possible candidates for the *k*th position of *a*, given the contents of the first *k-1* positions. The number of candidates returned in this array is denoted by ncandidates.
- `process_solution(a,k)`-This routine prints, counts, or somehow processes a complete solution once it is constructed.

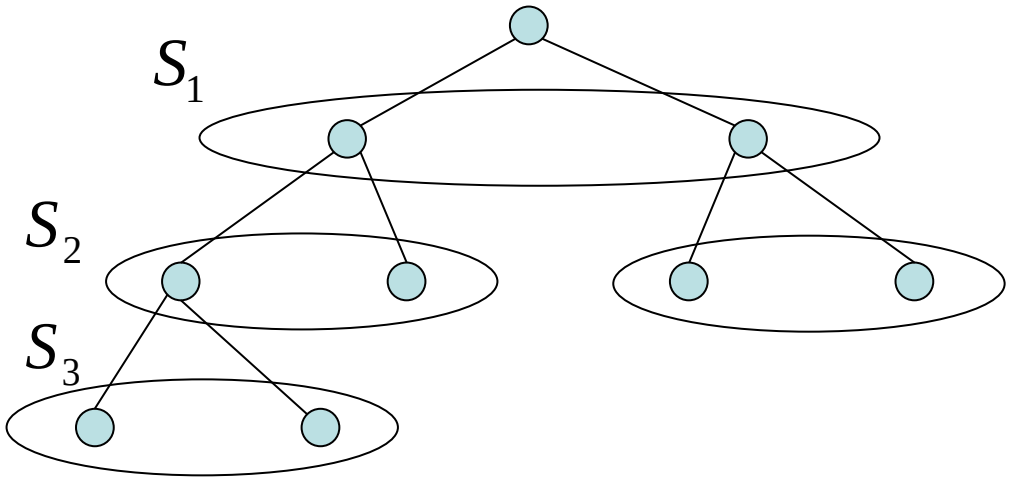
Application-Specific Routines (cont'd)

- Backtracking ensures correctness by enumerating all possibilities. It ensures efficiency by never visiting a state more than once.
- Because a new candidates array *c* is allocated with each recursive procedure call, the subsets of not-yet-considered extension candidates at each position will not interfere with each other.



Problem: Constructing All Subsets

- We can construct the 2^n subsets of n items by iterating through all possible 2^n length-n vectors of *true* or *false*, letting the ith element denote whether item i is or is not in the subset.
- Using the notation of the general backtrack algorithm, S_k=(*true*, *false*), and a is a solution whenever $k \geq n$.



Constructing All Subsets (cont'd)

```
main()
{
  int a[NMAX]; /* solution vector */

  backtrack(a,0,3);
}
```

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
    c[0] = TRUE;
    c[1] = FALSE;
    *ncandidates = 2;
}
```

```
process_solution(int a[], int k)
{
  int i;    /* counter */

  printf("{");
  for (i=1; i<=k; i++)
    if (a[i] == TRUE) printf(" %d",i);

  printf(" }\n");
}
```

```
is_a_solution(int a[], int k, int n)
{
  return (k == n);
}
```

```
bool finished = FALSE;          /* found all solutions yet? */

backtrack(int a[], int k, data input)
{
   int c[MAXCANDIDATES];  /* candidates for next position */
   int ncandidates;       /* next position candidate count */
   int i;                 /* counter */

   if (is_a_solution(a,k,input))
      process_solution(a,k,input);
   else {
      k = k+1;
      construct_candidates(a,k,input,c,&ncandidates);
      for (i=0; i<ncandidates; i++) {
         a[k] = c[i];
         backtrack(a,k,input);
         if (finished) return;  /* terminate early */
      }
   }
}
```

Results

{ 1 2 3}
{ 1 2}
{ 1 3}
{ 1}
{ 2 3}
{ 2}
{ 3}
{}

Constructing All Permutations

- To avoid repeating permutation elements, we must ensure that the *i*th element of the permutation is distinct from all the elements before it.
- To use the notation of the general backtrack algorithm, $S_k=\{1,2,\ldots,n\}-a$, and a is a solution whenever $k=n$:

Constructing All Permutations (cont'd)

```
main()
{
  int a[NMAX]; /* solution vector */
  backtrack(a,0,n);
}
```

```
is_a_solution(int a[], int k, int n)
{
  return (k == n);
}
```

```
process_solution(int a[], int k)
{
  int i;    /* counter */
  for (i=1; i<=k; i++) printf(" %d",a[i]);
  printf("\n");
}
```

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
  int i;                /* counter */
  bool in_perm[NMAX];   /* what is now in the permutation? */

  for (i=1; i<NMAX; i++) in_perm[i] = FALSE;
  for (i=1; i<k; i++) in_perm[ a[i] ] = TRUE;

  *ncandidates = 0;
  for (i=1; i<=n; i++)
    if (in_perm[i] == FALSE) {
      c[ *ncandidates] = i;
      *ncandidates = *ncandidates + 1;
    }
}
```

```
bool finished = FALSE;          /* found all solutions yet? */

backtrack(int a[], int k, data input)
{
   int c[MAXCANDIDATES];  /* candidates for next position */
   int ncandidates;       /* next position candidate count */
   int i;                 /* counter */

   if (is_a_solution(a,k,input))
      process_solution(a,k,input);
   else {
      k = k+1;
      construct_candidates(a,k,input,c,&ncandidates);
      for (i=0; i<ncandidates; i++) {
         a[k] = c[i];
         backtrack(a,k,input);
         if (finished) return;  /* terminate early */
      }
   }
}
```

Result

- When n=3 in the main program, the results look:

1 2 3

1 3 2

2 1 3

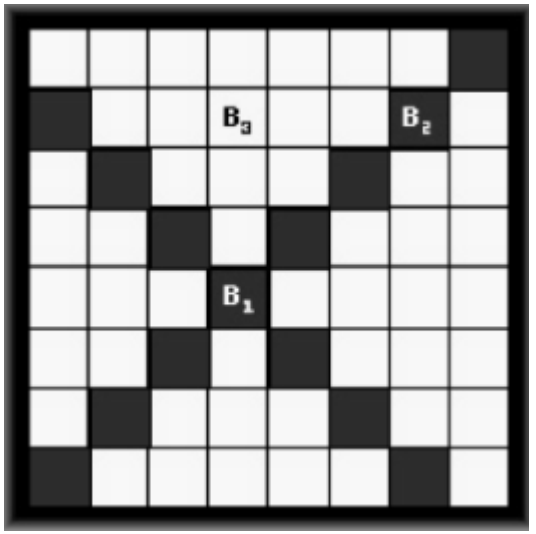
2 3 1

3 1 2

3 2 1

Prob: Little Bishops

- The Problem
  - A bishop is a piece used in the game of chess which can only move diagonally from its current position. Two bishops attack each other if one is on the path of the other. In the figure above, the dark squares represent the reachable locations for bishop B_1 from its current position. Bishops B_1 and B_2 are in attacking position, while B_1 and B_3 are not. Bishops B_2 and B_3 are also in non-attacking position.



Prob: Little Bishops (cont'd)

- The Problem
  - Given two numbers *n* and *k*, determine the number of ways one can put *k* bishops on an *n*x*n* chessboard so that no two of them are in attacking positions.
- The Input
  - The input file may contain multiple test cases. Each test case occupies a single line in the input file and contains two integers $n(1 \leq n \leq 8)$ and $k(0 \leq k \leq n^2)$.
  - A test case containing two zeros terminates the input.

Prob: Little Bishops (cont'd)

- The Output
  - For each test case, print a line containing the total number of ways one can put the given number of bishops on a chessboard of the given size so that no two of them lie in attacking positions. You may safely assume that this number will be less than 10^{15}.
- Sample Input

```
8 6
4 4
0 0
```

- Sample Output

```
5599888
260
```

Prob: Queue

- The Problem
  - Consider a queue with *N* people, each of a different height. A person can see out to the left of the queue if he or she is taller than all the people to the left; otherwise the view is blocked. Similarly, a person can see to the right if he or she is taller than all the people to the right.
  - A crime has been committed, where a person to the left of the queue has killed a person to the right of the queue using a boomerang. Exactly *P* members of the queue had unblocked vision to the left and exactly *R* members had unblocked vision to the right, thus serving as potential witnesses.
  - The defense has retained you to determine how many permutations of *N* people have this property for a given *P* and *R*.

Prob: Queue (cont'd)

- The Input
  - The input consists of T test cases, with T ($1 \leq T \leq 10{,}000$) given on the first line of the input file.
  - Each test case consists of a line containing three integers. The first integer N indicates the number of people in a queue ($1 \leq N \leq 13$). The second integer corresponds to the number of people who have unblocked vision to their left (P). The third integer corresponds to the number of people who have unblocked vision to their right (R).

Prob: Queue (cont'd)

- The Output
  - For each test case, print the number of permutations of *N* people where *P* people can see out to the left and *R* people can see out to the right.
- Sample Input

```
3
10 4 4
11 3 1
3 1 2
```

- Sample Output

```
90720
1026576
1
```

Prob: Tug of War

- The Problem
  - Tug of war is a contest of brute strength, where two teams of people pull in opposite directions on a rope. The team that succeeds in pulling the rope in their direction is declared the winner.
  - A tug of war is being arranged for the office picnic. The picnickers must be fairly divided into two teams. Every person must be on one team or the other, the number of people on the two teams must not differ by more than one, and the total weight of the people on each team should be as nearly equal as possible.

Prob: Tug of War (cont'd)

- The Input
  - The input begins with a single positive integer on a line by itself indicating the number of test cases following, each described below and followed by a blank line.
  - The first line of each case contains *n*, the number of people at the picnic. Each of the next *n* lines gives the weight of a person at the picnic, where each weight is an integer between 1 and 450. There are at most 100 people at the picnic.
  - Finally, there is a blank line between each two consecutive inputs.

Prob: Tug of War (cont'd)

- The Output
  - For each test case, your output will consist of a single line containing two numbers: the total weight of the people on one team, and the total weight of the people on the other team. If these numbers differ, give the smaller number first.
  - The output of each two consecutive cases will be separated by a blank line.

Prob: Tug of War (cont'd)

- Sample Input

```
1

3
100
90
200
```

- Sample Output

```
190 200
```