Data Structures and Algorithms

Balagopal Komarath

2025.07

Chapter 1

Brute-force search

Brute-force search refers to a class of algorithms that tries each candidate solution one-by-one. It is simple but inefficient. It usually appears as a subroutine to handle small cases within more efficient algorithms. It can also be used when input sizes are small enough.

We demonstrate brute-force search by counting the number of solutions to the n-queens problem. In the n-queens problem, the goal is to find a way, or to find all ways, or to count the number of ways to place n queens on an $n \times n$ chessboard such that no queen threatens another. The following is a solution for $n = 4$:

```
.q..
...q
q...
..q.
```

A brute-force search can solve this problem for small values of n. In a brute-force search, we try candidate solutions one-by-one. What is a candidate? The simplest definition is that it is a placement of n queens on an $n \times n$ board. For $n = 8$, this gives us $\binom{64}{8}$ possibilities. A computer can do roughly 10^8 operations per second. So going through all such candidates will take at least 45 seconds, assuming one operation per candidate, which is a conservative estimate. If we define candidates as those placements that has one queen per row, then the number of candidates is now only 8^8, which is 45 times less. We may also define candidates by a permutation of $[0, n)$, where the i^{th} element in the permutation is the column occupied by the queen on the i^{th} row. Then, there are only $8! < 8^8$ candidates. The way we define the candidates can make a huge difference in the running-time.

The following function `nqueens(n)` counts the number of solutions using brute-force search.

```
1 size_t nqueens(size_t n)
2 {
3   size_t c = 0;
4   size_t *queens = malloc(n * sizeof(size_t));
5   assert(queens);
6   first(n, queens);
7   while (true) {
```

```
 8     if (ok(n, queens))
 9       ++c;
10     if (!next(n, queens)) break;
11   }
12   return c;
13 }
```

Observe that in the above function, there is nothing specific to the n-queens problem. All the details of the problem are encapsulated into the following three functions:

- `first()`, for some fixed ordering of the candidates, writes the first candidate into the array `candidate`.
- `next()` modifies the `candidate` array so that it becomes the next candidate in the fixed ordering. It returns `true` if and only if there was a next candidate. If it returns `false`, then the value in `candidate` should not be used.
- `ok()` returns `true` if and only if the array `candidate` holds a valid solution.

We will now implement these functions for the n-queens problem. Recall that the `candidate` array stores a permutation of $[0, n)$. We will order these permutations lexicographically as follows (for $n = 3$): 012, 021, 102, 120, 201, 210. More formally, the ordering is:

Definition 1. *Let $p = (p_1, \ldots, p_n)$ and $q = (q_1, \ldots, q_n)$ be permutations of some set of n numbers. They are* lexicographically equal *if and only if $p_i = q_i$ for $1 \leq i \leq n$. We say that p is* lexicographically less than *q if and only if there is a $1 \leq j \leq n$ such that $p_i = q_i$ for all $i < j$ and $p_j < q_j$.*

The first permutation is the identity permutation:

```
1 void first(size_t n, size_t *queens)
2 {
3   for (size_t i = 0; i < n; ++i)
4     queens[i] = i;
5 }
```

To find the next permutation, we scan backwards from the end of the array until we see an element that is less than the previous one. Let the position of this element be i. Then, we swap this element with the rightmost element greater than it in the array. Finally, we reverse the sub-array starting from $i + 1$. For example:

```
next({1, 5, 2, 3, 0})
           ^  |
swap
->   {1, 5, 3, 2, 0}
reverse
->   {1, 5, 3, 0, 2}
```

Observe that if we cannot find an element that is less than the previous one while scanning backwards, then the elements are in decreasing order, which is the last permutation. Putting all this together, we obtain the following function:

```
1 bool next(size_t n, size_t *queens)
2 {
3   ssize_t i = n-2;
4   while (i >= 0 && queens[i] >= queens[i+1])
5     --i;
6   if (i < 0)
7     return false;
8   size_t j = n-1;
9   while (queens[j] <= queens[i])
10    --j;
11  swap(&queens[i], &queens[j]);
12  reverse(queens, i+1, n-1);
13  return true;
14 }
```

We now write the function to check validity of the candidate as a solution. By our definition of a candidate, two queens cannot be in the same row or same column. So for each queen, we just need to check whether there is any queen threatening it diagonally.

```
1 bool ok(size_t n, size_t *queens)
2 {
3   for (size_t i = 0; i < n; ++i)
4     for (size_t j = i+1; j < n; ++j) {
5       size_t d;
6       if (queens[i] < queens[j])
7         d = queens[j]-queens[i];
8       else
9         d = queens[i]-queens[j];
10      if (d == j-i)
11        return false;
12    }
13  return true;
14 }
```

Ex. 1 — What is the largest board size k for which the brute-force search algorithm finishes within five minutes? Determine the wall clock time taken by the `nqueens()` function for $n \in [2, k]$.

Ex. 2 — Modify the n-queens program to print the first solution. What is the largest board size k for which you can obtain a solution within five minutes? Determine the number of permutations looked at before finding a solution for $n \in [2, k]$.

Ex. 3 — Write a function:

```
bool le(size_t *p, size_t *q, size_t n);
```

that determines whether or not the permutation `p` is lexicographically less than or equal to `q`.

Ex. 4 — Write the function:

```
bool prev(size_t n, size_t *perm);
```

to modify the permutation in `perm` to the previous permutation and return `true` if possible and otherwise return `false`.

Ex. 5 — Define a candidate as any placement of one queen per row. Rewrite the n-queens program to use this definition. Do a performance comparison of both solutions.

Ex. 6 — Describe an algorithm that given two natural numbers n and i outputs the i^{th} permutation of $[0, n)$ in lexicographic order. This algorithm should finish within reasonable time for $n \leq 10^{10}$.

Ex. 7 — Describe an algorithm that given a natural number n and a permutation σ of $[0, n)$ returns the index of σ in lexicographic ordering.

Ex. 8 — An ordering of all permutations of $[0, n)$ is called a *Gray ordering* if two consecutive permutations in the ordering differ at exactly two places. For example, for $n = 3$:

```
012
021
120
102
201
210
```

is a Gray ordering. Do Gray orderings exist for all n? If so, design `first()` and `next()` functions to iterate over them. If not, argue why.

Ex. 9 — Why does `next()` correctly compute the next permutation?

Chapter 2

Backtracking

Backtracking improves brute-force search by introducing the concept of a *partial candidate* to reduce the number of candidates.

We have seen that brute-force search is too inefficient to solve n-queens for $n \geq 13$. We define a placement of queens on the first i rows as a partial candidate. Consider the partial candidate $0, 1$ for $n = 13$. There are 11! candidates that extend this partial candidate. However, we know that none of them can be a solution because the first two queens threaten each other.

Backtracking is best described using a combinatorial object called a *rooted, ordered tree*. A rooted, ordered tree (hereafter, just referred to as a tree) is a set V of *nodes* along the function $\mathsf{children} : V \mapsto V^*$, where $V^* = \cup_{i \geq 0} V^i$. For any $u \in V$ there is at most one $v \in V$ such that $\mathsf{children}(v)$ contains u and u appears exactly once in such a sequence. There is exactly one $r \in V$ such that it does not appear in the image of $\mathsf{children}$. This node r is called the *root* of the tree. A node u such that $\mathsf{children}(u) = ()$ is called a *leaf* of the tree.

For $n = 3$, we can define a tree of partial permutations as follows:

```
children(.)   = (0, 1, 2)
children(0)   = (01, 02)
children(1)   = (10, 12)
children(2)   = (20, 21)
children(01)  = (012)
children(02)  = (021)
children(10)  = (102)
children(12)  = (120)
children(20)  = (201)
children(21)  = (210)
children(012) = ()
children(021) = ()
children(102) = ()
children(120) = ()
children(201) = ()
children(210) = ()
```

It is more convenient to use the following representation for a tree of partial permutations of $[0, 3)$:

.

```
0
  01
    012
  02
    021
1
  10
    102
  12
    120
2
  20
    201
  21
    210
```

The root node is the empty permutation and is the only node in the first column. It has three children: 0, 1, and 2 that appear in the second column. The node 0 has two children 01 and 12 that appear in the third column and so on.

Backtracking will go through the nodes of the tree in the following order:

```
.
-> 0
-> 01
<- 0
-> 02
-> 021
<- 02
<- 0
<- .
-> 1
-> 10
<- 1
-> 12
<- 1
<- .
-> 2
-> 20
-> 201
<- 20
<- 2
-> 21
<- 2
<- .
```

The backtracking algorithm performs more work that brute-force search for small n such as 3 or 4. But for larger values, backtracking performs less work than brute-force search due to the ability to eliminate a large number of candidates without looking at them. The following function implements a backtracking algorithm to count the number of solutions to the n-queens puzzle:

```
 1 size_t nqueens(size_t n)
 2 {
 3   struct node p;
 4   root(n, &p);
 5   size_t c = 0;
 6   while (true) {
 7     if (leaf(&p)) {
 8       if (ok(&p))
 9         ++c;
10       if (!up(&p))
11         break;
12     } else {
13       if (!ok(&p)) {
14         if (!up(&p))
15           break;
16       } else {
17         if (!down(&p))
18           if (!up(&p))
19             break;
20       }
21     }
22   }
23   return c;
24 }
```

The function `ok()` returns `false` when the given partial candidate is known to be invalid. The functions `up()` and `down()` move up and down in the tree of partial candidates. They return `false` when this movement is not possible.

Lines 7 to 13 deal with candidate solutions. Lines 13 and 14 implement the backtracking optimization. Observe that failure to move up a tree in lines 10, 14, and 18 indicates that no candidates are left.

The `struct node` represents a node in the tree of partial candidates. The field `k` is the number of elements in the partial permutation. Note that storing only the partial permutation is insufficient. Consider the partial permutation 1 for $n = 3$. Calling `down()` two times on this node produced the candidates 10 and 12. So the node should also remember the last child that was visited from this node on a `down()` call. The field `last` keeps track of this information. It stores the last element of the partial permutation of the last visited child or `-1` if no such child exists.

```
1 struct node {
2   size_t n;
3   size_t *queens;
4   size_t k;
5   ssize_t last;
6 };
```

To go up from a node, we have to reduce `k` by one and set `last` to the removed element.

```
1 bool up(struct node *p)
2 {
```

```
3   if (p->k == 0)
4     return false;
5   p->last = p->queens[p->k-1];
6   --(p->k);
7   return true;
8 }
```

To move down in the tree, we extend the permutation by one more element. This element should be the smallest element greater than `last` that does not appear elsewhere in the partial permutation.

```
1 bool down(struct node *p)
2 {
3   bool *used = malloc(p->n * sizeof(bool));
4   assert(used);
5   for (size_t i = 0; i < p->n; ++i)
6     used[i] = false;
7   for (size_t i = 0; i < p->k; ++i)
8     used[p->queens[i]] = true;
9
10  for (size_t i = p->last+1; i < p->n; ++i)
11    if (!used[i]) {
12      p->queens[p->k] = i;
13      ++(p->k);
14      p->last = -1;
15      free(used);
16      return true;
17    }
18  free(used);
19  return false;
20 }
```

Lines 3 to 8 determine which elements are already used in the permutation. The `for` loop in line 10 searches for the smallest element larger than `last` that we can use to extend the partial permutation. If there is no such element, then we must have come up to this node from the last child and so we return `false` in line 19.

Ex. 10 — What is the largest board size k for which the backtracking algorithm can solve n-queens within five minutes? Compare these times with that of the brute-force search algorithm.

Ex. 11 — For both brute-force search and backtracking algorithms, estimate the largest board size for which these algorithms would find the number of solutions within one year from your running-time measurements.

Ex. 12 — Modify the backtracking algorithm to print the first solution instead. What is the largest board size k for which the backtracking algorithm can find a solution to the n-queens puzzle within five minutes? Compare these times with that of the brute-force search algorithm.

Ex. 13 — Write a SUDOKU solver that uses backtracking search to find a solution.

Ex. 14 — A Knight's closed tour on a chessboard is a sequence of moves of a Knight that starts and ends at the same square and occupies each square exactly once. Write a program to print a solution given n, m, and the starting square for an $n \times m$ board.

Chapter 3

Analysis

So far, we have used wall clock time to measure the performance of our algorithms. This is the most direct measure of efficiency. However, it could differ from one machine to another. It also depends on the other programs running on the machine at the time of measurement, called the context.

We can devise measurements to estimate performance that are independent of the machine and context. For the brute-force search, a good measure is the number of times `next()` is called. For the backtracking search, the number of times `up()` and `down()` are called may be used to estimate performance. More generally, we have to determine what to measure after understanding the program.

Ex. 15 — Measure the above parameters and compare the performance of brute-force search and backtracking. Compare these measurements with actual wall clock time measurements.

Chapter 4

Recursion

Recursion in programming is a technique where we use the function we are writing in the body of the function. The use may be direct, where the function body contains a call to the same function, or indirect, where the function body calls another function that calls this function. This technique quite often yields to simpler, but sometimes slightly inefficient, solutions.

The following `min()` function uses direct recursion.

```
 1 int min(const int *a, size_t n)
 2 {
 3   assert(n >= 1);
 4   if (n == 1)
 5     return a[0];
 6   int rest = min(a+1, n-1);
 7   if (a[0] < rest)
 8     return a[0];
 9   return rest;
10 }
```

Lines 4 and 5 solves what is called a *base case*. Base cases must be solved without invoking the same function (directly or indirectly). Line 6 to 9 implement the *recursive case*.

The following code uses indirect recursion to implement functions to check whether a natural number is even or odd:

```
1 bool even(int n)
2 {
3   return n == 0 || odd(n-1);
4 }
5
6 bool odd(int n)
7 {
8   return n == 1 || even(n-1);
9 }
```

The above functions are almost as easy to implement without recursion. We will now see a recursive implementation of a function that prints all k-permutations of a set of n elements.

```
1 void do_permutations (
2   int *a, size_t n,
3   int *b, size_t k,
4   size_t k1
5 )
6 {
7   if (k1 == k) {
8     print(b, k);
9     return;
10  }
11
12  for (size_t i = 0; i < n; ++i) {
13    swap(&a[0], &a[i]);
14    b[k1] = a[0];
15    do_permutations(a+1, n-1, b, k, k1+1);
16    swap(&a[0], &a[i]);
17  }
18 }
19
20 void permutations (
21   int *a, size_t n,
22   int *b, size_t k
23 )
24 {
25   do_permutations(a, n, b, k, 0);
26 }
```

To generate all k-permutations, we call the function with the following arguments: The array `a` is a set of `n` elements and the array `b` should have space for storing at least k elements. We then call the helper function `do_permutations()` to do the actual work. In the `do_permutations()` function, the argument `k1` should be the number of elements in `b` so far. Lines 7 to 10 is the base case. At this point, we have built one permutation and simply print the array and return. Lines 12 to 17 is the recursive case. We put each element of `a` as the next element in the permutation and append each one of the $(k-1)$-permutations of the other elements in `a` to this element. Since in an array it is difficult to remove an element that is not at the first position, we swap the element to be removed with the first element and then remove the first element.

The following depicts the tree of function calls made to generate all 2-permutations of `{0, 1, 2}`. We use `p` instead of `do_permutations()`. The 2-permutations occurring at the leaves of the tree are printed.

```
1 p({0, 1, 2}, 3, {}, 2, 0)
2   p({1, 2}, 2, {0}, 2, 1)
3     p({2}, 1, {0, 1}, 2, 2)
4     p({1}, 1, {0, 2}, 2, 2)
5   p({0, 2}, 2, {1}, 2, 1)
6     p({2}, 1, {1, 0}, 2, 1)
7     p({0}, 1, {1, 2}, 2, 2)
8   p({1, 0}, 2, {2}, 2, 1)
9     p({0}, 1, {2, 1}, 2, 2)
```

```
10      p({1}, 1, {2, 0}, 2, 2)
```

A recursive function to generate all k-combinations of an n-element set is given below:

```
 1 void do_combinations (
 2   const int *a, size_t n,
 3   int *b, size_t k,
 4   size_t k1
 5 )
 6 {
 7   if (n < k-k1)
 8     return;
 9
10   if (k1 == k) {
11     print(b, k);
12     return;
13   }
14
15   b[k1] = a[0];
16   do_combinations(a+1, n-1, b, k, k1+1);
17   do_combinations(a+1, n-1, b, k, k1);
18 }
19
20 void combinations (
21   const int *a, size_t n,
22   int *b, size_t k
23 )
24 {
25   do_combinations(a, n, b, k, 0);
26 }
```

It uses the following recursive formulation: A k-element set can be constructed by adding the first element to each $(k-1)$-element set from the rest of the elements, or by considering all k-element subsets from the rest of the elements. This formulation also gives rise to the following combinatorial identity:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \text{where } n > k, k > 0.$$

We can implement backtracking search for n-queens using recursion. Observe that the recursive generation of permutations builds them by starting with the empty permutation and recursively extending them. So we can use it with minimal modifications.

```
1 size_t do_nqueens (
2   size_t n, size_t *queens, size_t k
3 )
4 {
5   if (k == n && ok(queens, k))
6     return 1;
7
```

```
 8   if (!ok(queens, k))
 9     return 0;
10
11   bool *used = malloc(n * sizeof(bool));
12   assert(used);
13   for (size_t i = 0; i < n; ++i)
14     used[i] = false;
15   for (size_t i = 0; i < k; ++i)
16     used[queens[i]] = true;
17
18   size_t c = 0;
19   for (size_t i = 0; i < n; ++i)
20     if (!used[i]) {
21       queens[k] = i;
22       c += do_nqueens(n, queens, k+1);
23     }
24   free(used);
25   return c;
26 }
27
28 size_t nqueens(size_t n)
29 {
30   size_t *queens = malloc(n * sizeof(size_t));
31   assert(queens);
32   size_t c = do_nqueens(n, queens, 0);
33   free(queens);
34   return c;
35 }
```

Observe that lines 8 and 9 implement backtracking. The `return` statements move up the tree and the recursive call in line 22 move down the tree.

Ex. 16 — Observe that the recursive function to generate permutations does not generate them in lexicographic order. Write a recursive function to generate permutations in lexicographic order.

Ex. 17 — Show the tree of calls made to generate all 2-combinations of `{0, 1, 2, 3}` by the `do_combinations()` function.

Ex. 18 — Show the tree of calls made by `nqueens()` for $n = 4$.

Ex. 19 — Measure the wall clock time for the recursive, backtracking n-queens solution and compare it with the non-recursive implementation.

Ex. 20 — Write a recursive function to generate all k-permutations with replacement for a given n-element set.

Ex. 21 — Write a recursive function to generate all k-combinations with replacement for a given n-element set.

Ex. 22 — Write a recursive function to generate all partitions with exactly k parts for a given n-element set.

Chapter 5

Searching

The linear search algorithm searches for an element in a sequence of elements. In the following code, we represent the sequence using an array of elements called `haystack`. The element we are looking for is `needle`.

```
 1 template <typename T>
 2 T* search (
 3   T* haystack, size_t n,
 4   const T& needle
 5 )
 6 {
 7   for (size_t i = 0; i < n; ++i)
 8     if (haystack[i] == needle)
 9       return &haystack[i];
10   return nullptr;
11 }
```

Observe that the code is *generic*. That is, it works for any type `T`. The only requirement from this type is that it should allow us to check for equality of elements using the operator `==` (See line 8).

Since the algorithm is generic, the wall clock running time will depend on the time taken to compare two elements. We wish to avoid this dependence. So we assume comparison takes some unit time and measure with respect to that unit. Instead of a concrete wall clock time, we study how the performance scales with the input size. Linear search has the following performance characteristics:

Best Case The needle is the first element of the haystack. The running time is independent of the size of the haystack. In other words, the running time is bounded by a constant.

Worst Case The needle is not in the haystack. The running time is linear in the size of the haystack.

Average Case The analysis depends on the distribution of the elements in the haystack. If we assume that all these are drawn uniformly at random from $[1, N]$ for some N, then the search will terminate at the first element with probability $1/N$, at the second element with probability $(1 - 1/N)1/N$ and so on. Let the random Variable X denote the number of elements.

Then:

$$P(X = i) = (1 - t)^i t$$

where $t = 1/N$ and:

$$E[X] = t \sum_{i=1}^{n} i(1 - t)^i$$

which can be analyzed using the formula for arithmetico geometric series for various values of N.

Linear search only assumes that elements can be compared for equality. If we impose the stronger requirement on inputs that elements of the type are ordered, then we can use a much more efficient algorithm called binary search. We also need the elements in `haystack` to be arranged in that order.

```
template <typename T>
T *search (
  T *haystack, size_t n,
  const T& needle
)
{
  size_t begin = 0, end = n;

  while (begin < end) {
    size_t mid = begin + (end - begin) / 2;
    if (haystack[mid] < needle)
      begin = mid + 1;
    else if (haystack[mid] > needle)
      end = mid;
    else
      return &haystack[mid];
  }
  return nullptr;
}
```

Here's how the search works on an example array while looking for a `6`. The `^` marker indicates the element compared with the needle. The size of the sub-array to be searched thereafter is roughly halved after each comparison. We show the values of `begin` and `end` in the last two columns.

```
1 2 4 5 9 10 | 0 6
      ^
        9 10 | 4 6
          ^
        9    | 4 5
        ^
             | 4 4
```

Ex. 23 — Analyze the average case running time for linear search in the following cases:

1. $N = 10$.
2. $N = \sqrt{n}$.

3.$N = n$.

4.$N = n^2$.

5.$N = 2^n$.

Check the results of your analysis by measuring running times.

Ex. 24 — Analyze the running time of Binary search in the best, worst, and average cases.

Ex. 25 — Let i be the index of a needle in the haystack or N if it is absent from the haystack. Design an algorithm that runs in time proportional to $\log(i)$ to search for the needle in the haystack if the elements are arranged in order.

Chapter 6

Asymptotic Notation

Asymptotic notation is a machine independent way to denote the performance of algorithms. We define the following set parameterized by functions on natural numbers:

$$O(f) = \{g \mid \text{for some } c \text{ and } n_0\ g(n) \leq cf(n) \text{ for } n \geq n_0\}$$

where $f, g : \mathbb{N} \mapsto \mathbb{N}$, $c \in \mathbb{Q}$, and $n_0 \in \mathbb{N}$.

Let us analyze binary search in the worst case. Let k be the maximum amount of time taken by a single iteration of the main loop. Then, the running time $T(n)$ spent by the main loop on a haystack of size n is:

$$T(n) \leq T(\lceil n/2 \rceil) + k$$

when $n > 1$ and $T(0) \leq T(1) \leq k$.

Observing $T(n)$ for different values of n, we see:

n	$T(n)$
1	k
2	$2k$
3	$3k$
4	$3k$
$\vdots$	$\vdots$
8	$4k$
$\vdots$	$\vdots$
16	5k

The interesting cases are when n is a power of two. Observe $T(n)$ is k, $2k$, $3k$, ..., for n in 1, 2, 4, ... For the values of n not a power of two, the value $T(n)$ is bounded by its value at the next power of two. We can formalize this using a proof by induction and obtain $T \in O(\log)$. This is usually written as $T(n) = O(\log n)$.

Proof. We split the proof into two cases and show that in both cases $T(n) = O(\log n)$.

We first consider the case when n is a power of two. Say $n = 2^p$, then we claim $T(n) \leq k(p + 1)$. The base case has $p = 0$ and we know $T(1) \leq k$. For the inductive case, we have $T(2^p) \leq T(2^{p-1}) + k \leq kp + k = k(p + 1)$.

When n is not a power of two, we have a p such that $2^p \leq n < 2^{p+1}$. The running time can be bounded as $T(n) \leq T(2^{p+1}) \leq k(p+2) \leq (k+1)\log n$ for $n \geq 2^{2k}$. □

We also need to show that in the worst case, binary search needs at least $\log n$ time. This cannot be done using $O(.)$ notation as it only provides an upper bound. We define another parameterized set:

$$\Omega(f) = \{g \mid \text{for some } c \text{ and } n_0 \ f(n) \geq g(n) \text{ for } n \geq n_0\}$$

A worst case input for binary search is when the needle is absent from the haystack. Let k be the minimum amount of time taken by one iteration of the loop. We observe $T(1) \geq k$, $T(2) \geq 2k$, ... As before, we can use induction to prove $T(n) = \Omega(\log n)$.

We also have the set $\Theta(f) = O(f) \cap \Omega(f)$ that provides a single notation for both upper and lower bounds for a function on natural numbers.

Ex. 26 — What is the best case running time for binary search in asymptotic notation?

Ex. 27 — Show that the average case running time for linear search is $T(n) = \Theta(n)$.

Ex. 28 — Suppose $f(n) \leq kg(n) + c$ for some constants k, c, and n_0 for all $n \geq n_0$. Prove that $f = O(g)$.

Ex. 29 — Prove: If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.

Ex. 30 — Prove: If $f = g_1 + g_2$ and $g_1, g_2 \in O(h)$, then $f = O(h)$.

Ex. 31 — Prove: If $f = g_1 g_2$, $g_1 \in O(h_1)$, and $g_2 \in O(h_2)$, then $f = O(h_1 h_2)$.

Chapter 7

Sorting

Chapter 8

Graphs

A graph is an abstract mathematical object that can model many real-world problems. Mathematically, a graph is a set V along with a relation $E \subseteq V \times V$. If the set V is finite, the graph is called a *finite graph*. An element $v \in V$ is called a *vertex* or a *node* and an element $e = (u, v) \in E$ is called an *edge* or an *arc* with the vertex u as its *tail* and v as its *head*. We sometimes say e is an edge from u to v to specify the direction. An edge (u, u) is called a *self-loop*. A graph without self-loops is called *simple*. A graph is called *undirected* if for all $u, v \in V$, we have $(u, v) \in E$ if and only if $(v, u) \in E$. We say that u and v are *adjacent* if $(u, v) \in E$ in an undirected graph.

We will now see how to model some problems as problems on graphs. Consider the undirected graph where V corresponds to the cells on a chessboard and (u, v) is an edge if and only if a u and v are in the same row, column, or diagonal. The problem of placing n non-threatening queens on the chessboard is then the same as finding a subset $U \subseteq V$ such that $|U| = n$ and there are no edges between vertices in U. Such a subset is called an *independent set* of the graph.

Consider the undirected graph where V corresponds to the cells in a SUDOKU puzzle. Two vertices are adjacent if and only if they are in the same row, column, or box. Some vertices are assigned a color which is a number from 1 to 9 such that adjacent vertices do not have the same color. The SUDOKU puzzle is then to extend this coloring to all vertices while preserving the property that adjacent vertices do not have the same color. The general version of this problem is called the *graph coloring* problem.

Chapter 9

Depth-first Search

Chapter 10

Breadth-first Search

Chapter 11

Shortest Paths

Chapter 12

Minimum Spanning Trees

Chapter 13

Topological Sorting

Chapter 14

Hash Tables

Chapter 15

Binary Search Trees

Chapter 16

Balanced Binary Search Trees

Chapter 17

Full Source Code for Examples

Brute-force Search for n-queens

```
#include <assert.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void swap(size_t *a, size_t *b)
{
  size_t t = *a;
  *a = *b;
  *b = t;
}

void reverse(size_t *a, size_t b, size_t e)
{
  while (b < e) {
    swap(&a[b], &a[e]);
    ++b;
    --e;
  }
}

void first(size_t n, size_t *queens)
{
  for (size_t i = 0; i < n; ++i)
    queens[i] = i;
}

bool next(size_t n, size_t *queens)
{
  ssize_t i = n-2;
  while (i >= 0 && queens[i] >= queens[i+1])
    --i;
```

```
35   if (i < 0)
36     return false;
37   size_t j = n-1;
38   while (queens[j] <= queens[i])
39     --j;
40   swap(&queens[i], &queens[j]);
41   reverse(queens, i+1, n-1);
42   return true;
43 }
44
45 bool ok(size_t n, size_t *queens)
46 {
47   for (size_t i = 0; i < n; ++i)
48     for (size_t j = i+1; j < n; ++j) {
49       size_t d;
50       if (queens[i] < queens[j])
51         d = queens[j]-queens[i];
52       else
53         d = queens[i]-queens[j];
54       if (d == j-i)
55         return false;
56     }
57   return true;
58 }
59
60 size_t nqueens(size_t n)
61 {
62   size_t c = 0;
63   size_t *queens = malloc(n * sizeof(size_t));
64   assert(queens);
65   first(n, queens);
66   while (true) {
67     if (ok(n, queens))
68       ++c;
69     if (!next(n, queens))
70       break;
71   }
72   return c;
73 }
74
75 int main(int argc, const char *argv[])
76 {
77   size_t n = 8;
78   if (argc > 1) {
79     n = 0;
80     const char *arg = argv[1];
81     while (*arg)
82       n = n * 10 + (*arg++ - '0');
83   }
84   printf("%ld\n", nqueens(n));
```

```
85   return 0;
86 }
```

Backtracking Search for n-queens

```
 1 #include <assert.h>
 2 #include <stdbool.h>
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
 6 struct node {
 7   size_t n;
 8   size_t *queens;
 9   size_t k;
10   ssize_t last;
11 };
12
13 void root(size_t n, struct node *p)
14 {
15   p->n = n;
16   p->k = 0;
17   p->last = -1;
18   p->queens = malloc(n * sizeof(size_t));
19   assert(p->queens);
20 }
21
22 bool ok(const struct node *p)
23 {
24   for (size_t i = 0; i < p->k; ++i)
25     for (size_t j = i+1; j < p->k; ++j) {
26       size_t d;
27       if (p->queens[i] < p->queens[j])
28         d = p->queens[j]-p->queens[i];
29       else
30         d = p->queens[i]-p->queens[j];
31       if (d == j-i)
32         return false;
33     }
34   return true;
35 }
36
37 bool up(struct node *p)
38 {
39   if (p->k == 0)
40     return false;
41   p->last = p->queens[p->k-1];
42   --(p->k);
43   return true;
44 }
45
46 bool down(struct node *p)
```

```
47 {
48   bool *used = malloc(p->n * sizeof(bool));
49   assert(used);
50   for (size_t i = 0; i < p->n; ++i)
51     used[i] = false;
52   for (size_t i = 0; i < p->k; ++i)
53     used[p->queens[i]] = true;
54
55   for (size_t i = p->last+1; i < p->n; ++i)
56     if (!used[i]) {
57       p->queens[p->k] = i;
58       ++(p->k);
59       p->last = -1;
60       free(used);
61       return true;
62     }
63   free(used);
64   return false;
65 }
66
67 bool leaf(const struct node *p)
68 {
69   return p->k == p->n;
70 }
71
72 size_t nqueens(size_t n)
73 {
74   struct node p;
75   root(n, &p);
76   size_t c = 0;
77   while (true) {
78     if (leaf(&p)) {
79       if (ok(&p))
80         ++c;
81       if (!up(&p))
82         break;
83     } else {
84       if (!ok(&p)) {
85         if (!up(&p))
86           break;
87       } else {
88         if (!down(&p))
89           if (!up(&p))
90             break;
91       }
92     }
93   }
94   return c;
95 }
96
```

```
 97 int main(int argc, const char *argv[])
 98 {
 99   size_t n = 8;
100   if (argc > 1) {
101     n = 0;
102     const char *arg = argv[1];
103     while (*arg)
104       n = n * 10 + (*arg++ - '0');
105   }
106   printf("%ld\n", nqueens(n));
107   return 0;
108 }
```

Recursive Generation of All Permutations

```
 1 #include <stdio.h>
 2
 3 void print(const int *a, size_t n)
 4 {
 5   printf("<␣");
 6   for (size_t i = 0; i < n; ++i)
 7     printf("%d␣", a[i]);
 8   printf(">\n");
 9 }
10
11 void swap(int *a, int *b)
12 {
13   int t = *a;
14   *a = *b;
15   *b = t;
16 }
17
18 void do_permutations (
19   int *a, size_t n,
20   int *b, size_t k,
21   size_t k1
22 )
23 {
24   if (k1 == k) {
25     print(b, k);
26     return;
27   }
28
29   for (size_t i = 0; i < n; ++i) {
30     swap(&a[0], &a[i]);
31     b[k1] = a[0];
32     do_permutations(a+1, n-1, b, k, k1+1);
33     swap(&a[0], &a[i]);
34   }
35 }
36
```

```
37 void permutations (
38   int *a, size_t n,
39   int *b, size_t k
40 )
41 {
42   do_permutations(a, n, b, k, 0);
43 }
44
45 int main()
46 {
47   int a[] = {0, 1, 2, 3};
48   int b[2] = {0};
49
50   permutations(a, 4, b, 2);
51
52   return 0;
53 }
```

Recursive Generation of All Combinations

```
 1 #include <stdio.h>
 2
 3 void print(const int *a, size_t n)
 4 {
 5   printf("<␣");
 6   for (size_t i = 0; i < n; ++i)
 7     printf("%d␣", a[i]);
 8   printf(">\n");
 9 }
10
11 void do_combinations (
12   const int *a, size_t n,
13   int *b, size_t k,
14   size_t k1
15 )
16 {
17   if (n < k-k1)
18     return;
19
20   if (k1 == k) {
21     print(b, k);
22     return;
23   }
24
25   b[k1] = a[0];
26   do_combinations(a+1, n-1, b, k, k1+1);
27   do_combinations(a+1, n-1, b, k, k1);
28 }
29
30 void combinations (
31   const int *a, size_t n,
```

```
32   int *b, size_t k
33 )
34 {
35   do_combinations(a, n, b, k, 0);
36 }
37
38 int main()
39 {
40   const int a[] = {0, 1, 2, 3};
41   int b[2] = {0};
42
43   combinations(a, 4, b, 2);
44
45   return 0;
46 }
```

Recursive Backtracking Search for n-queens

```
 1 #include <assert.h>
 2 #include <stdbool.h>
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
 6 bool ok(const size_t *queens, size_t k)
 7 {
 8   for (size_t i = 0; i < k; ++i)
 9     for (size_t j = i+1; j < k; ++j) {
10       size_t d;
11       if (queens[i] < queens[j])
12         d = queens[j] - queens[i];
13       else
14         d = queens[i] - queens[j];
15       if (d == j-i)
16         return false;
17     }
18   return true;
19 }
20
21 size_t do_nqueens (
22   size_t n, size_t *queens, size_t k
23 )
24 {
25   if (k == n && ok(queens, k))
26     return 1;
27
28   if (!ok(queens, k))
29     return 0;
30
31   bool *used = malloc(n * sizeof(bool));
32   assert(used);
33   for (size_t i = 0; i < n; ++i)
```

```
34      used[i] = false;
35    for (size_t i = 0; i < k; ++i)
36      used[queens[i]] = true;
37
38    size_t c = 0;
39    for (size_t i = 0; i < n; ++i)
40      if (!used[i]) {
41        queens[k] = i;
42        c += do_nqueens(n, queens, k+1);
43      }
44    free(used);
45    return c;
46  }
47
48  size_t nqueens(size_t n)
49  {
50    size_t *queens = malloc(n * sizeof(size_t));
51    assert(queens);
52    size_t c = do_nqueens(n, queens, 0);
53    free(queens);
54    return c;
55  }
56
57  int main(int argc, const char *argv[])
58  {
59    size_t n = 8;
60    if (argc > 1) {
61      n = 0;
62      const char *arg = argv[1];
63      while (*arg)
64        n = n * 10 + (*arg++ - '0');
65    }
66    printf("%ld\n", nqueens(n));
67    return 0;
68  }
```

Generic Linear Search

```
1  #include <cassert>
2  #include <iostream>
3
4  template <typename T>
5  T* search (
6    T* haystack, size_t n,
7    const T& needle
8  )
9  {
10   for (size_t i = 0; i < n; ++i)
11     if (haystack[i] == needle)
12       return &haystack[i];
13   return nullptr;
```

```
}

int main()
{
  int haystack[] = { 5, 9, 1, 4, 2, 10 };
  assert(search(haystack, 6, 4));
  assert(!search(haystack, 6, 11));
  return 0;
}
```

Binary Search

```
#include <cassert>
#include <iostream>

template <typename T>
T *search (
  T *haystack, size_t n,
  const T& needle
)
{
  size_t begin = 0, end = n;

  while (begin < end) {
    size_t mid = begin + (end - begin) / 2;
    if (haystack[mid] < needle)
      begin = mid + 1;
    else if (haystack[mid] > needle)
      end = mid;
    else
      return &haystack[mid];
  }
  return nullptr;
}

int main()
{
  int haystack[] = { 1, 2, 4, 5, 9, 10 };
  assert(search(haystack, 6, 4));
  assert(!search(haystack, 6, 11));
  return 0;
}
```

Chapter 18

Compiling and Running Programs

This chapter lists some common commands used for compiling and running C/C++ programs.

To compile a program in `a.c` into an executable in file `a` for testing:

```
gcc -std=c23 -Wall -Wpedantic -g a.c -o a
```

To compile a program in `a.c` into an executable in file `a` for fast execution:

```
gcc -std=c23 -Wall -Wpedantic -O3 a.c -o a
```

For C++:

```
g++ -std=c++23 <same as gcc options>
```