Complete search

Generating subsets

Subsets of {0,1,2} are {}, {0}, {1}, {2}, {0,1}, {0,2}, {1,2} and {0,1,2}.

- bit manipulation of integers, to 64 elements.
- recursion

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

Generating permutations

Permutations of {0,1,2} are (0,1,2), (0,2,1), (1,0,2), (1,2,0), (2,0,1) and (2,1,0).

- recursion
- std::next_permutation

Backtracking

Begin empty, slowly add solutions (prune the search). Iterate through all the ways with optimizations.

- recursion

Meet in the middle

Clever observation that problem can be solved quicker if we divide in half (?) and merge halfs with another algorithm. Example: choose elements from [2,4,5,9] such that their sum x.

Greedy algorithms

Construct solution by always choosing the best solution at the moment, never looking back.

- Pros: are efficiency
- Cons: hard to proove that they are correct.

What works:

- euro coins work, but general coins not necessarily
- scheduling
    - start and end time, choose as many events

    - duration and deadline, be as little late as possible with deadlines
- minimizing sums (we know a1, a2, ..., find we must find x)
    - |a1 − x| + |a2 − x| + ··· + |an − x|
    - (a1 − x)^2 + (a2 − x)^2 + ··· + (an − x)^2
- data compression - Huffman coding

Dynamic programming

If we can divide original problem into overlapping subproblems and have a base case(s).

- finding an optimal solution
- counting the number of solutions

implementations:

- recursive (don't need to solve everything, but higher constant factors)
- iterative (solve every problem untill n with a for loop)

Use memoization with recursion! For loop always uses it.

Coin problem

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0) {
            value[x] = min(value[x], value[x-c]+1);
        }
    }
}
```

Constructing the solution (find the one optimal solution and show its construction from smaller problems)

- store the first chosen coin for *x*
- after the loop, read *y*. Now we can produce the path of getting *y* by getting the first coin from the stored array of chosen coins
- print the coin and than substract it from *y*
- repeat until *y* is 0

Counting the number of solutions

- instead of having *value* array, we can have a *count* array
- `count[x] += count[x-c];`

Longest increasing subsequence

Array from left to right, each element ==in the sequence== larger (ignore smaller).

- dynamic programming
- sort cloned array and do "longest common subsequence" problem with sorted array

Paths in a grid

Travel from upper left to bottom right, each square has a different score. You can move down and right. Get highest score.

Observe the subproblem "do we move to the current square from the upper or left side?"

```
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        sum[y][x] = max(sum[y][x-1],sum[y-1][x])+value[y][x];
    }
}
```

Knapsack problems

Receive a set of objects, find subset with some property.

Example:

Weights are [1,3,3,5]. Which sums can we construct (weights are consumed). Solution is an array of booleans.

`possible(x,k) = possible(x−wk,k −1)  possible(x,k −1)`

x ... needed sum

k ... first k weights

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = 0; x <= W; x++) {
        if (x-w[k] >= 0) // check for out of bounds
            possible[x][k] |= possible[x-w[k]][k-1];
        possible[x][k] |= possible[x][k-1];
    }
}
```

There also exists a solution with 1D array.

Edit distance

The minimum number of operations needed to convert one string to another. Allowed operations are:

- insert -> `distance(a, b − 1) + 1`
- remove -> `distance(a - 1, b −1) + 1`
- modify -> `distance(a - 1, b −1) + cost(a, b)`

```
if x[a] = y[b]
    cost(a,b) = 0
else
    cost(a,b) = 1
```

A table can show operations clearly.

Counting tilings

Calculate the number of distinct ways to fill an n × m grid using 1×2 and 2×1.

- formula (!)
- dynamic programming (the state of the current row only depends on previous ==row== [singular])

Each row iterate through every possible option and discard incompatible ones. Optimization: squash unique characters that behave the same.

Amortized analysis

Used for an alternative to big O notation (more realistic).

Two pointers method

- subarray sum: O(n) -> subarray [a, b] such that contents sum up to sum *x*
- 2SUM and 3SUM: just two or three elements sum up to *x*, needs sorting

Nearest smaller elements

Estimate number of operations on a data structure, even if uneven. Total number must be limited.

- for each element find the first smaller element that preceds it
- stack structure O(n)

Sliding window minimum

- constant size window
- for every position, calculate something
  - minimum or maximum
- queue structure O(n)

Range queries

Calculate something for a subarray between a range.

- sum
- min
- max

```
int sum(int a, int b) {
    int s = 0;
    for (int i = a; i <= b; i++) {
        s += array[i];
    }
    return s;
}
```

Static array queries

Array are never updated between the queries. Also works higher dimension arrays.

Sum queries -> prefix sum array

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2
1	4	8	16	22	23	27	29

define `sum(0, −1) = 0`

`sum(a, b) = sum(0, b) − sum(0, a − 1)`

Min and max queries

Efficiently get the minimum or maximum value in a range in O(1). Prepocessing is O(n logn).

Precalculate `min(a, b) where b − a + 1` (the length of the range) is a power of two

```
min(a, b) = min(min(a, a + w − 1), min(a + w, b)),
w = (b − a + 1) / 2
```

Get the value:

```
min(a, b) = min(min(a, a + k − 1), min(b − k + 1, b))
```

Binary indexed tree

Variant of prefix sum array.

Supports:

- processing a range sum query
- updating a value

Allows us to efficiently update array values between sum queries. With prefix sum array you need to rebuild it after every update.

It's not a tree, it's an array.

`p(k) denotes the largest power of two that divides k`

```
tree[k] = sum(k − p(k) + 1, k)
```

Each position k contains the sum of values in a range of the original array whose length is p(k) and that ends at position k

- O(logn) for query

Segment tree

Other queries

- minimum
- maximum
- greatest common divisor
- bit operations and, or and xor

Supports:

- processing a range query
- updating an array value
- sum queries, minimum and maximum and many other operations
- O(log n) for both operations

VS. binary tree:

- more general +
- more memory -
- more difficult to implement -

Additional techniques

Index compressions

Use a hash map to map original indexes to compressed indexes. When indexes are too large 10^9

```
c(8) = 1
c(555) = 2
c(109) = 3
```

Range updates

When we want the reverse of range queries. Update ranges and retrieve single values.

We build a difference array. Values indicate the differences between consecutive values in the original array. The original array is the prefix sum array of the difference array.

0	1	2	3	4	5	6	7
3	3	1	1	1	5	2	2
3	0	-2	0	0	4	-3	0

Update a range in the original array by changing just two elements in the difference array. A more difficult problem is to support both range queries and range updates.

Bit manipulation

- a signed number $-x$ equals an unsigned number $2n - x$
- x << k appends k zero bits
- x >> k removes the k last bits
- x | (1 << k) sets the kth bit of x to one
- x & ~(1 << k) sets the kth bit of x to zero
- x ^ (1 << k) inverts the kth bit of x
- x & (x−1) sets the last one bit of x to zero

- x | (x−1) inverts all the bits after the last one bit
- x is a power of two exactly when x & (x−1) = 0

`the kth bit of a number is one exactly when x & (1 << k) is not zero`

Prints the binary representation of a number.

```
for (int i = 31; i >= 0; i--) {
    if (x&(1<<i)) cout << "1";
    else cout << "0";
}
```

- __builtin_clz(x): the number of zeros at the beginning of the number
- __builtin_ctz(x): the number of zeros at the end of the number
- __builtin_popcount(x): the number of ones in the number
- __builtin_parity(x): the parity (even or odd) of the number of ones

There are also long long versions of the functions available with the suffix ll.

Representing sets

Every subset of a set `{0,1,2,...,n−1}` can be represented as an n bit integer whose one bits indicate which elements belong to the subset.

- efficient way to represent sets
- one bit of memory per element
- set operations are bit operations

```
/* Create a set */
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4: the length of set

/* Print the set */
for (int i = 0; i < 32; i++) {
    if (x&(1<<i))
        cout << i << " ";
}
// output: 1 3 4 8
```

	set syntax	**bit syntax**
intersection	a ∩ b	a & b
union	a ∪ b	a | b
complement	$\overline{a}$	~a
difference	a \ b	a & (~b)

The following code goes through the subsets of `{0,1,...,n−1}`:

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

The following code goes through the subsets with exactly *k* elements:

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

The following code goes through the subsets of a set *x*:

```
int b = 0;
do {
    // process subset b
} while (b=(b-x)&x);
```

Bit optimizations

- algorithms can be optimized with bit operations
- same time complexity, but faster execution

Hamming distances

The **Hamming distance** `hamming(a,b)` between two strings a and b of equal length is the number of positions where the strings differ.

`hamming(01101,11001) = 2`

Counting subgrids

Given an n× n grid whose each square is either black (1) or white (0), calculate the number of subgrids whose all corners are black.

`color[y][x] denotes the color in row y and column x`

O(n^3)

```
int count = 0;
for (int i = 0; i < n; i++) {
    if (color[a][i] == 1 && color[b][i] == 1)
        count++;
}
```

O(n^3/N), where N is the number of bits (int or ll)

```
int count = 0;
for (int i = 0; i <= n/N; i++) {
    count += __builtin_popcount(color[a][i]&color[b][i]);
}
```

Dynamic programming

states can be stored as integers, which is efficient

Optimal selection

Example:

- we are given the prices of k products over n days, and we want to buy each product exactly once
- however, we are allowed to buy at most one product in a day
- what is the minimum total price?

`price[x][d]` denotes the price of product *x* on day *d*

`total(S, d)` denotes the minimum total price for buying a subset S of products by day d

`total({}, d) = 0` it doesn't cost anything to buy an empty set

`total({x}, 0) = price[x][0]` one way to buy one product on the first day

`total(S, d) = min(total(S, d − 1), min(total(S \ x, d − 1) + price[x][d]))`

This means that we either do not buy any product on day d or buy a product x that belongs to S. In the latter case, we remove x from S and add the price of x to the total price.

From permutations to subsets

- permutations: n!
- subsets: 2^n

Example: There is an elevator with maximum weight *x*, and *n* people with known weights who want to get from the ground floor to the top floor. **What is the minimum number of rides needed if the people enter the elevator in an optimal order?**

x = 10, n = 5

person	weight
0	2
1	3
2	3
3	5
4	6

O(n! * n) permutatuins -> O(2^n * n) dynamic programming

Counting subsets

For every subset S there is an integer assigned to that particular set. The task is to calculate the sum of all those integers.

- value[{}] = 3
- value[{0}] = 1
- value[{1}] = 4

- `value[{0,1}] = 5`
- `value[{2}] = 5`
- `value[{0,2}] = 1`
- `value[{1,2}] = 3`
- `value[{0,1,2}] = 3`

`sum({0,2}) = value[{}] + value[{0}] + value[{2}] + value[{0,2}] = 3 + 1 + 5 + 1 = 10`

O(2^(2n)) go through all pairs of subsets -> O(2^n * n) dynamic programming