08-1. 비트 연산

기본적인 비트 연산

- BOJ 20360 Binary numbers: 양의 정수 n이 주어지면 n을 이진법으로 나타내었을 때 켜진 비트의 위치를 구하는 문제
  - `for(int i=0; i<20; i++) if(n & (1 << i)) cout << i << " ";`
  - `for(int i=0; i<20; i++) if(n >> i & 1) cout << i << " ";`
  - `for(int i=0; i<20; i++, n>>=1) if(n & 1) cout << i << " ";`
- BOJ 1182 부분수열의 합: 크기가 $n \leq 20$인 중복 집합이 주어지면, 합이 s인 부분 집합의 수를 구하는 문제
  - 부분 집합은 비트로 표현할 수 있음
  - (0-based 기준) a_i가 포함되면 2^i를 나타내는 비트를 켜는 방식
  - 모든 부분 집합을 0 이상 $2^n - 1$ 이하의 정수에 일대일 대응시킬 수 있음
    - 공집합은 0, 모든 원소를 포함하는 집합은 $2^n - 1$
  - 따라서 1 이상 $2^n - 1$ 이하의 모든 정수를 차례로 보면서, 켜진 비트에 대응되는 수들의 합을 구하면 됨
  - http://boj.kr/835a3d7cedf948fb846f404cbb3ff587
  - 모든 2^n가지의 부분 집합을 보면서, 매번 $O(n)$ 시간에 합을 구하므로 시간 복잡도는 $O(n2^n)$.
- BOJ 11723 집합: 1 이상 20 이하의 수를 원소로 하는 집합에 대한 연산을 처리하는 문제
  - add x: `S |= 1 << x;`
  - remove x: `S &= ~(1 << x);`
  - check x: `cout << S >> x & 1 << "\n";`
  - toggle x: `S ^= 1 << x;`
  - all: `S = (1 << 21) - 2`
  - empty: `S = 0`

조금 더 복잡한 비트 연산

- 2의 보수
  - 컴퓨터에서 음수를 표현하는 방법
  - `-x = (~x) + 1`, 모든 비트를 반전한 다음 1을 더함
- 이진법으로 나타냈을 때 마지막 1의 위치(lowest set bit)를 구하는 방법
  - `lower_set_bit(x) = x & -x = x & ((~x)+1)`
  - `x`와 `~x`는 모든 위치에서 서로 다르기 때문에 `x & ~x = 0`
  - `~x`에 1을 더하면 `~x`의 가장 아래에 있는 연속한 1들이 모두 0으로 바뀌고, 바로 뒤에 있는 0이 1로 바뀜
    - `01001111` -> `01010000`
    - 즉, `x`의 가장 아래에 있는 0들에 대응되는 자리는 0으로 바뀌고, 바로 다음에 있는 1(가장 뒤에 있는 1)에 대응되는 자리가 1로 바뀜
  - 따라서 `x`와 `-x`에서 모두 1인 위치는 가장 뒤에 있는 1밖에 없음
  - 이 방법을 이용하면 BOJ 1182 부분수열의 합 문제를 $O(2^n)$에 해결할 수 있음
  - http://boj.kr/426e0b86e8394ff39f6249c17ee7e607

- 모든 submask를 순회하는 방법
  - `for(int msk=bit; msk; msk=bit&(msk-1))` 에서 `msk`는 `bit`의 모든 submask를 내림차순으로 순회함
  - `msk-1`은 `msk`에서 가장 밑에 있는 켜진 비트를 끄고, 그 밑에 있는 모든 비트를 켬
  - 그 값을 `bit`와 and 해서 실제로 bit에서 켜져 있는 비트만 남기는 방식
  - `for(int bit=0; bit<(1<<n); bit++) for(int msk=bit; msk; msk=bit&(msk-1))`의 시간 복잡도는 $O(3^n)$
    - $(1+2)^n$와 이항 계수의 관계를 생각해 보자.

08-2. C++ 관련

`std::sort`와 `std::stable_sort`

- `std::sort`
  - 일반적으로는 퀵 정렬이 매우 빠르게 동작하지만, 재귀 깊이가 깊어지면 $O(n^2)$이 될 수 있음
  - 또한, 배열의 크기가 충분히 작으면 삽입 정렬이 가장 빠름
  - 따라서 `std::sort`는 일단 퀵 정렬을 수행하되, 깊이가 `2 * floor(log_2 n)`보다 깊어지면 $O(n \log n)$을 보장하기 위해 힙 정렬 수행
  - 만약 퀵 정렬 단계에서 분할된 배열의 크기가 16 이하면 더 정렬을 수행하지 않고 그냥 놔둠
  - 모든 과정이 끝나면 대부분 정렬되어 있지만, 크기가 16 이하인 몇몇 덩어리 내부는 정렬되지 않은 상태
  - 이러한 배열에서 삽입 정렬을 사용하면 각 원소가 최대 16칸만 이동하므로 $O(n)$에 나머지 부분을 정렬할 수 있음
  - 최악의 경우에도 $O(n \log n)$을 보장함
- `std::stable_sort`
  - `std::stable_sort`는 합병 정렬로 구현되어 있음
  - 일반적으로 원소 간의 비교 횟수는 `std::sort`보다 `std::stable_sort`가 더 적음
  - 하지만 `std::sort`가 더 하드웨어 친화적이기 때문에 비교 연산과 교환의 수행 시간이 비슷하면 보통 `std::sort`가 더 빠름
  - 비교 연산이 상수 시간이 아니라면 `std::stable_sort`가 더 빠를 수 있음

`std::vector` 관련

- `std::vector`의 구현 방법
  - 미리 적당한 공간을 할당해 놓음
  - `push_back`을 이용해 원소를 추가할 때는 미리 할당받은 공간에 차곡차곡 쌓음
  - 만약 할당받은 공간을 전부 다 사용했다면 기존 공간의 2배 크기의 새로운 공간을 할당받아서, 지금까지 갖고 있던 모든 원소를 새로운 공간으로 이동
  - 한 번에 삽입에서는 $O(n)$ 만큼의 시간이 걸릴 수 있지만, n번의 삽입에서 필요한 연산을 모두 더하면 $O(n)$이 됨을 알 수 있음
    - $1+2+4+\cdots+2^k = 2^{k+1}-1$이기 때문
- `std::vector` 사용 시 주의점
  - 이 코드의 실행 결과는 `5 2 3 4 5`일 것 같지만 `1 2 3 4 5`가 출력됨
  - `new_element` 함수에서 재할당이 발생해서 `v[0]`의 위치가 바뀌는데, `v[0] = new_element(5);`에서 좌변을 먼저 평가해서 발생하는 일

        - `v[0]`의 주소를 레지스터에 넣고, `new_element`를 실행한 다음에, 그 결과를 기존에 기존에 구한 `v[0]`의 주소에 넣기 때문
    - 구체적으로, C++14까지는 `b = a`에서 `a`, `b`의 평가 순서가 정해지지 않은 상태였음
    - C++17부터는 정해짐: 참고
    - 위 코드를 컴파일할 때 -std=c++17 옵션을 넣으면 `5 2 3 4 5`가 출력됨을 확인할 수 있음
    - 어셈블리 코드도 확인해 보자.
        - -std=c++14 컴파일: `vector::operator[]`를 호출한 다음에 `new_element`를 호출함
        - -std=c++17 컴파일: `new_element`를 호출한 다음에 `vector::operator[]`를 호출함

range-based for문 관련

- `for(auto i : "abcd")`를 하면 안 되는 이유
    - `string s = "abcd"; for(auto i : s) cout << (int)i << " ";`의 출력 결과는 `97 98 99 100`
    - 하지만 `for(auto i : "abcd") cout << (int)i << " ";`의 출력 결과는 `97 98 99 100 0`
    - `"abcd"`의 타입은 `std::string`이 아니라 `const char *`이기 때문에 뒤에 NULL이 붙음
    - `for(auto i : string("abcd")) cout << (int)i << " ";`를 사용하자.
- range-based for문을 사용할 수 있을 조건
    - `begin(a)`와 `end(a)`가 정의되어 있으면 `for(auto i : a)`를 사용할 수 있음
    - `for(auto it=begin(a); it!=end(a); ++it){ i = *it; }`와 같은 방식으로 작동하기 때문
    - C-style array는 begin과 end가 정의되어 있으므로 사용 가능
        - `template<typename T, size_t S> T* begin(T (&a)[S]){ return a; }`
        - `template<typename T, size_t S> T* end(T (&a)[S]){ return a + S; }`
        - 이런 식으로 타입, 크기에 대해 템플릿 함수가 정의되어 있기 때문
    - 동적으로 할당받은 배열을 사용 불가능
- `std::array` 관련
    - (실제로는 조금 다르지만) `std::array`는 `template<typename T, size_t S> class array{ T a[S]; };`처럼 구현되어 있음
    - 두 배열을 swap할 때 배열의 길이에 비례하는 시간이 걸림
    - 반면, `std::vector, std::deque, std::set, std::map` 등은 swap할 때 상수 시간

08-3. 하계(lower bound) 관련

- 비교 기반 정렬 하한이 $\Omega(n \log n)$인 이유
    - 적절한 비교를 이용해 n개의 원소를 나열하는 $n!$가지의 방법 중 하나를 선택해야 함
    - 비교 결과에 따라 행동하는 것은 이진 트리 형태의 결정 트리로 나타낼 수 있음
    - 결정 트리의 높이 하한이 비교 기반 정렬의 하한이 될 텐데, 리프 정점이 $n!$가지이므로 높이의 하한은 $\Omega(\log n!)$
    - $\frac{n}{2}\log\frac{n}{2} \leq \log n! \leq n \log n$이므로 $\Omega(n \log n)$
- 정렬된 배열에서 원하는 값(key)을 찾을 때 이분 탐색이 최적인 이유
    - "상대자"라는 개념을 도입함
    - key가 존재하는 위치가 처음에 정해진 것이 아닌, 악의를 가진 "상대자"가 비교 횟수가 최대가 되도록 모순이 생기지 않는 선에서 위치를 유동적으로 결정
    - 만약 중간 지점이 아닌 다른 지점을 선택하면, 상대자는 양쪽 중 더 큰 구간으로 정답을 배치할 수 있음 → 이분 탐색보다 더 안 좋아짐

    - 따라서 이분 탐색은 최악의 경우에서 비교 횟수를 최소화함
- $3 \times \lceil n/2 \rceil - 2$번의 비교로 최댓값과 최솟값을 모두 찾는 방법
    - 단순히 최댓값과 최솟값을 찾으면 $2 \times (n-1)$번의 비교가 필요함
    - 원소를 2개씩 묶어서 비교하자. (A_1 v.s. A_2, A_3 v.s. A_4, $\cdots$, A_{N-1} v.s. A_N) - $\lfloor n/2 \rfloor$번의 비교
    - 더 큰 값은 위로 올리고 작은 값은 아래로 내리면 최댓값은 위에 있는 원소 중 하나, 최솟값은 아래에 있는 원소 중 하나
    - 따라서 위에 있는 $\lceil n/2 \rceil$개의 원소들의 최댓값을 $\lceil n/2 \rceil - 1$번의 비교로 구할 수 있음
    - 마찬가지로 아래에 있는 $\lceil n/2 \rceil$개의 원소들의 최솟값을 $\lceil n/2 \rceil - 1$번의 비교로 구할 수 있음
- $n + \lceil \log_2 n \rceil - 2$번의 비교로 두 번째로 큰 값을 찾는 방법
    - 최댓값을 구하는 토너먼트를 만들면 $n-1$번의 비교로 가장 큰 값을 찾을 수 있음
    - 두 번째로 큰 값은 최댓값을 제외한 다른 원소한테 지지 않음
    - 따라서 두 번째로 큰 값은 최댓값에게 진 $\lceil \log_2 n \rceil$개의 원소 중 하나
    - 그러므로 $\lceil \log_2 n \rceil - 1$번의 비교로 두 번째로 큰 값을 찾을 수 있음

08-4. 기타

- Barrett reduction(코드), 나머지 연산을 할 때 제수가 상수면 뺄셈, 곱셈, 시프트 연산을 이용해 나머지를 구할 수 있음
    - $10^9 + 7$ 등의 수를 `constexpr int MOD = 1e9+7;` 처럼 컴파일 타임 상수로 선언하면 컴파일러가 알아서 최적화함
    - 컴파일러가 사용하는 방식은 barrett reduction과는 조금 다른 방식
- IEEE 754, 부동 소수점 자료형의 구현 방식
- 실수의 실수, 부동 소수점의 오차 관련
    - `double`은 $[-2^{52}, 2^{52})$ 구간을 벗어난 정수를 정확하게 표현하지 못할 수 있음
    - `cout << (1e18 == 1e18 + 1);`은 1을 출력함