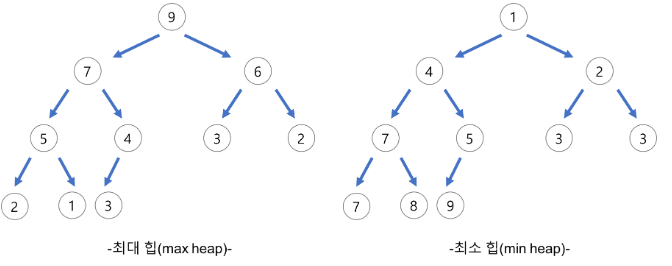
**4) "힙정렬(heapify), 우선순위 큐",**

힙정렬 : 힙은 완전 이진 트리의 일종으로 우선 순위 큐를 위하여 만들어진 자료구조로 최댓값, 최솟값을 쉽게 추출할 수 있는 자료구조이다.

최대 힙 트리나 최소 힙 트리를 구성해 정렬을 하는 방법으로 내림차순 정렬을 위해서는 최대 힙을 구성하고 오름차순 정렬을 위해서는 최소 힙을 구성하면 된다.

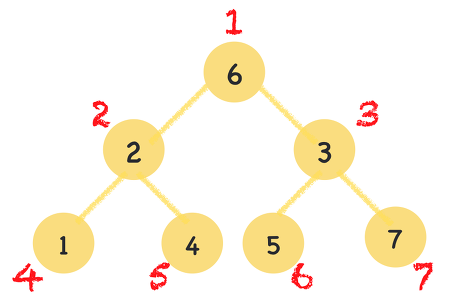


**- 최대 힙의 삽입 과정**  
1. N개의 노드에 대한 완전 이진 트리를 구성한다. 이때 루트 노드부터 부모 노드, 왼쪽 자식 노드 오른쪽 자식 노드 순으로 구성한다.   
2. 가장 마지막 위치에 이어서 새로운 요소를 삽입  
3. 부모 노드와 비교하면서 부모 노드보다 크다면 교환을 반복. O(log2N)

**- 최대 힙의 삭제 과정**  
1. 최대 힙에서 최댓값은 루트 노드이므로 루트 노드가 삭제된다.  
2. 삭제된 루트 노드에는 힙의 마지막 노드를 가져온다.  
3. 힙을 재구성한다. (내 자식 노드와 비교하면서 교환을 반복한다) O(log2N)

**힙 정렬의 장점**  
- 시간 복잡도가 좋은편 O(nlog2n)  
- 가장 유용한 경우는 전체 자료 정렬이 아닌 가장 큰 값 몇개만 필요할 경우  
- 추가적인 배열이 필요하지 않아 메모리 효율적

**힙 정렬의 단점**  
- 단순히 속도만 가지고 비교하면 퀵정렬이 평균적으로 더 빠름

**힙 정렬의 구현**  
  
현재 노드가 i라고 가정했을 때, 부모 노드의 인덱스는 i/2   
왼쪽 자식 노드의 인덱스 i\*2, 오른쪽 자식 노드의 인덱스 i\*2+1

1. 최대 힙의 구현 (오름 차순) <https://dpdpwl.tistory.com/45>

|  |
| --- |
| void heapify(int i) //i : 부모  {  int cur = 2 \* i; //자식 노드  //자식 노드 중 큰 값을 cur로  if(cur < n && heap[cur] < heap[cur+1]) cur++;  if(heap[i] < heap[cur])  {  swap(heap[i],heap[cur]); //부모와 자식 교환 시  if(cur <= n/2) heapify(cur); //자식 노드가 자식이 있으면 재귀  }  }  void heapsort(int i) //마지막 노드 부터 --  {  swap(heap[1],heap[i]);  int root = 1;  int cur = 2;  while(cur/2<i)  {  cur = 2\*root;  if(cur < i-1 && heap[cur] < heap[cur+1]) cur++;  if(cur < i && heap[root] < heap[cur])  swap(heap[root],heap[cur]);  root = cur;  }  }  int main(){  scanf("%d",&n);  for(int i = 1; i <= n; i++)  scanf("%d",&heap[i]);  for(int i = n/2; i > 0; i--) // 최초 heap 생성  heapify(i);  for(int i = n; i > 0; i--) // heap 정렬  heapsort(i);  for(int j = 1; j <= n; j++) // 출력  printf("%d ",heap[j]);  } |

**7) "DP, 피보,동전"(18 하 DS)**

동적 프로그래밍(Dynamic Programming)

: 작은 부분 문제들을 모두 해결한 후에 그 해들을 이용하여 보다 큰 크기의 부분 문제들을 해결하여 최종적으로 원래 주어진 문제를 해결하는 알고리즘, 메모이제이션 기법을 미리 작은 부분 문제들의 해를 저장해두고 반복적인 계산을 하지 않는다.

⇒ 분할정복도 작은 문제를 해결한 후 합치는 것 아닌가요?

: 분할정복에서는 겹치는 문제가 발생하지 않지만 DP에서는 겹치는 문제가 발생하고 이를 해결하기 위해 메모이제이션 기법을 활용하는 것입니다.

크게 두가지 방식으로 나뉜다. Top-down 하향식 / Bottom-up 상향식

피보나치 수열 f(n) = f(n-1) + f(n-2) //유사한 문제로 팩토리얼 문제

f(1) = 0, f(2) = 1  
구현 방법이 3가지.

1. 재귀 O(2^n)
2. DP(top-down) O(n)
3. DP 반복문(bottom-up) O(n)

1. 재귀

|  |
| --- |
| int fibo(int n){  if(n <= 1) //기저조건  return n;  return fibo(n-1) + fibo(n-2);  } |

2. 동적 프로그래밍(top-down)

|  |
| --- |
| int fibo(int n){  if(fibo\_memo[n] != -1)  return fibo\_memo[n];  fibo\_memo[n] = fibo(n-1) + fibo(n-2);  return fibo\_memo[n];  } |

3. 반복문

|  |
| --- |
| int fibo(int n){  arr[0] = 0;  arr[1] = 1;  for(int i=2; i<=n; i++){  arr[i] = arr[i-1] + arr[i-2];  }  return fibo[n];  } |

동적 프로그래밍이나 반복문은 memoization 기법을 이용하기 때문에 추가적인 메모리를 소모하지만 효율적이다.

재귀는

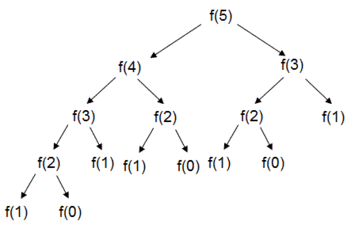
f(5)의 값을 구할 때 f(4) + f(3) 인데 앞서 사용했던 값을 기억하지 못하기 때문에

f(5) = f(3) + f(2) + f(2) + f(1) 로 나누어지며

f(5) = f(2) + f(1) + f(1)+ f(0) + f(1) + f(0) + f(1)

f(5) = f(1) + f(0) + f(1) + f(1)+ f(0) + f(1) + f(0) + f(1)

이렇게 계속해서 같은 값이 다시 쓰이기 때문에 비효율적이다.



**동전 문제**

만약 동전의 액수가 서로 배수 관계가 성립한다면 그리디 알고리즘으로 해결할 수 있다.

하지만 그렇지 않은 경우 DP를 이용해야 정확한 답이 나온다.

동전의 종류가 n개, 나타내고 싶은 가치 k

coin(0,k)로 시작해서 재귀(top-down)호출로 구현할 수 있다.

|  |
| --- |
| int coin(int n, int k){  if(n == N) return (k==0 ? 0 : IMPOSSIBLE); // base case  if(dp[n][k] != -1) return dp[n][k]; // 이미 계산됨    int result = coin(n+1, k);  if(k >= cost[n]) result = min(result, coin(n, k-cost[n]) + 1);  dp[n][k] = result; // dp 배열 값 갱신  return result;  } |

현재 동전을 사용하지 않고 다음 동전을 사용해서 k원을 구하는 경우와

현재 동전을 사용하고 f(n, k - 현재 동전의 가치) + 1(현재 동전 하나 사용)) 중 최솟값을 dp 배열에 저장하고 계속해서 사용하면 n == N , 즉 모든 동전을 사용했을 때 k=0 이면 해결할 수 있는 것이고 k!=0이면 해결할 수 없는 값이다.

같은 문제의 bottom up 방식

|  |
| --- |
| #include <cstdio>  #include <algorithm>  using namespace std;  const int IMPOSSIBLE = 1000000000;    int main(){  int N, K, cost[MAX\_N], dp[101][10001];  scanf("%d %d", &N, &K);  for(int i=0; i<N; i++)  scanf("%d", cost+i);  // dp 배열 초기화  for(int i=0; i<=N; i++)  for(int j=0; j<=K; j++)  dp[i][j] = IMPOSSIBLE;  // dp로 문제 품  for(int i=0; i<N; i++){  dp[i][0] = 0;  for(int j=0; j<=K; j++){  dp[i+1][j] = min(dp[i+1][j], dp[i][j]);  int jj = j + cost[i];  if(jj <= K) dp[i][jj] = min(dp[i][jj], dp[i][j] + 1);  }  }  // 정답 출력  if(dp[N-1][K] == IMPOSSIBLE) puts("-1");  else printf("%d\n", dp[N-1][K]);  } |

시간 복잡도는 O(nk)