3. 정보코드와 정보표현

2010 데이터로 표현하는 세상 요약본
고려대학교 김현철 교수
hkim64@gmail.com

코드가 맞는 사람

우리의 머릿속의 어떠한 정보를 밖으로 표현해 내어 다른 사람과 의사소통을 하려면, 어떠한 약속된(혹은 합의된) 기호의 형태로 표현을 해내야 한다. 그 표현 규칙을 '코드'라고 하고 그 코드를 이용하여 정보를 표현하는 것을 인코드(encode), 코드화된 정보를 다시 해석하는 과정을 디코드(decode)라고 한다.

결국 내 머릿속의 생각, 정보를 어떻게 효율적으로 표현할 것인가에 대한 것인데, 우리 인간이 사용하는 코드는 어떤 것들이 있을까. 언어, 문자, 숫자, 이모티콘 기호 같은 것들이 그 예가 될 수 있을 것이다.

먼저 숫자를 생각해보자. 먼 옛날의 누군가가 사냥해온 동물의 개수를 센다고 해보자. 무엇을 사용하여 그 숫자정보를 어떻게 표현하고 기록할 것인가. 인간은 숫자 정보를 표현하기 위하여 자연스럽게 '손가락'을 사용했다. 왜냐면 손가락은 폈다 접었다를 쉽게 할 수 있으므로 '표시'하기 수월 하기 때문이다. 그러면 우리는 일반적으로 10개의 손가락을 가지고 있으므로, 한꺼번에 10가지의 개수를 표시할 수 있게 되는 것이다. 즉, 양손을 합쳐서 하나의 표현 단위가 되는 것이고 그 표현 단위는 (펴는 손가락의 개수에 따라서) 10가지의 정보를 표현할 수 있다. (사실 손가락을 전혀 펴지 않은 상태, 즉 0도 표현한다면 모두 11개를 표현할 수 있다고 하지만 '0'의 개념은 후에 등장 했고, 숫자의 표현에 갯수 혹은 위치 혹은 자리수를 사용하게 되면서 오늘날 같은 10집법 체계가 갖추어 졌다) 즉 하나의 단위(즉, 한자리)에 10가지의 정보를 표현할 수 있는 것이다. 만약 발(손)가락이 양쪽 합쳐서 8개만 있는 독수리나 닭 같은 조류가 숫자를 만들었다면 아마도 8진법으로 만들었을 것이다. 두개의 발굽만 가지고 있는 돼지는 4진수를 사용했을까, 8진수를 사용했을까? 우리가 사용하는 10진수에서 처럼, 이렇게 생긴 모양을 이 정보로 표현하기로 하자라고 약속을 정해 놓고 사용하게 되는데 그 약속 규칙을 코드라고 한다. 자 그럼, 다시 이 10개의 손가락으로 또 어떤 다른 코드, 즉 정보를 표현하는 방법은 없을까?

방법1) 아까 해본 것처럼 모든 손가락을 접은 상태에서 숫자만큼 같은 개수의 손가락을 펴게되면 모두 10가지의 정보를 표현 할 수 있는 규칙을, 즉 코드를 만들 수 있게 된다. 이것은 10개의 손가락 그룹 전체를 하나의 표현 단위로 보고, 그 하나의 단위가 10개의 다른 정보를 표현하게 하는 것이다. 하지만, 표현할 수 있는 정보의 양이 매우 작아서 비효율적이다.

방법2) 이번에는 정보를 표현하는 단위가 하나의 손가락 각각 이라고 생각해보자. 그러면 이 단위에서는 손가락을 폈다, 접었다, 이 두 가지 정보만 표현할 수 있다. 하지만 우리는 이러한 표현 단위가 10개가 있으므로 (10개의 손가락!), 그 10개의 단위를 조합하면, 즉 10개의 손가락으로 만들 수 있는 다른 모양의 조합의 개수를 생각해보자. 하나의 손가락은 폈다 접었다만 하므로 2가지 정보를 표현할 수 있고 그런 것이 10개가 모여 있으니까 2x2x2x2x2x2x2x2x2x2=2^{10}, 즉 1024가지의 다른 정보를 표현할 수 있는 코드를 만들 수 있는 것이다. 0부터1023까지이 숫자를 열 손가락의 접고 펴고의 모양 조합 각각에 배정시켜 놓으면 그것이 바로 코드가 되는 것이고, 그 코드에 의해 숫자를 표현하는 것을 인코딩, 그 표현된 것을 다시 코드에 의해 해석하는 것은 디코딩 한다고 말한다. 이 방법은 우리가 사용하는 '이진수' 코딩 방법과 동일하다.

방법3) 만약 손가락을 접고 펴고만 할 수 있을 뿐만 아니라 중간 구부리기를 할 수 있다고 한다면, 손가락 하나로 3가지 정보를 표현할 수 있는 것이고, 그런 손가락이 (즉, 표현 단위가) 10개가 있으므로 그 10개가 표현해 낼 수 있는 모양의 조합의 개수는 3x3x3x3x3x3x3x3x3x3 = 3^{10} = 59,049가 된다. 놀랍지 않은가. 우리 손가락 10개로 거의 6만개의 정보를 표현할 수 있다니! 이것이 3진법이 되는 것이고, 방법2는 2진법이 되는 것이고, 방법1은 열 손가락을 하나의 단위로 생각해서 10가지의 정보를 표현한 것이니까 10진법이 되는 것이다. 즉 진법은 하나의 표현단위가 표현할 수 있는 정보의 개수에 의해서 정해지며, 단위의 개수가 늘어나면 표현할 수 있는 정보의 가짓수도 지수단위로 늘어나게 된다.

자 이번엔 조금 더 나가서, 우리의 언어를 생각해보자. 우리 한국사람이 (한국어 언어에서) 만들어 낼 수 있는 소리가 몇 가지나 될까. 그것이 무한의 수일까, 유한의 수일까. 아마도 유한일 것이다. 하지만 우리의 생각은 무한이다. 문제는, 유한개의 소리로 무한개의 정보를 표현해야 한다는 것이다. 다른 나라 사람들은 우리와는 다른 소리를 만들어 냄으로써 다른 표현 방법 (즉, 코드)를 사용한다. 우리가 사용하는 문자도 마찬가지이다. 한글을 보면 정교하고 분석적인 코드 체계를 갖추고 있다. 하나의 글자는 자음-모음-자음을 조합하여 만들어지게 된다. 자음은 표준자음은 14개(ㄱㄴㄷㄹㅁㅂㅅㅇㅈㅊㅋㅌㅍㅎ)이지만 이중자음(ㄲ ㄸ ㅃ ㅆ ㅉ)까지 포함하면 19개, 모음은 표준모음은 10개(ㅏㅑㅓㅕㅗㅛㅜㅠㅡㅣ)이지만 이중모음(ㅐ ㅒ ㅔ ㅖ ㅘ ㅙ ㅚ ㅝ ㅞ ㅟ ㅢ)까지 표현하면 21개가 된다. 그리고 받침에 사용되는 자음은 일반 자음 14개, 겹받침 자음 13개와 받침이 없는 경우를 합치면 모두 28개가 되어, 그들을 조합하면 모두 19x21x28 = 11,172개의 글자를 만드는 것이 이론적으로 가능하다. (사실 실제로 사용하는 글자는 이것보다 훨씬 작다. 사용되는 글자의 99.9%가 2300여개의 글자라고 한다.) 우리는 우리의 입으로 1만개가 넘는 다른 소리를 만들어 낼 수 있는가? 사실 우리 한국어에서 사용되는 모든 소리는 이것만 가지고도 충분하고 남을 것이다. 그럼에도 그 코드가 인간이 낼 수 있는 모든 소리를 다 표현 하는가. 영어 'F' 소리를 표현하는가. 영어 'R'과 'L'의 소리를 구분하여 표시하는가. 남아프리카 원주민의 그 처음 들어본 발음을 정확하게 표현하는가. 사실 우리나라 사람이 그러한 소리를 내지 않으니까 글자로도 표현되지 않는 것이지, 만약 그 새로운 소리를 어느 특정한 글자에 사용하기로 합의한다면 그것도 가능한 것이다. 사실 우리 한글 체계에서 사용하는 기호의 조합만 가지고 만들어 낼 수 있는 글자의 수는 (비록 사용은 안하지만) 더욱 많다. 한번 상상을 해보자. 표준자음이 14개 이므로 겹자음을 모두 억지로 만들어 낸다면 14x14=196가지 이므로 모두 210가지가 가능하다. 받침으로 사용되는 경우에는 받침이 없는 경우도 있으므로 211가지. 겹모음은 기본모음 10가지에서 2개를 골라내는 조합이므로 C(10,2)=45가지, 따라서 가능한 모음의 개수는 10+45=55가지. 그래서 가능한 전체 글자 기호의 개수는 210x55x211 = 2,437,050가 된다. 새로운 소리를 만들어 낸다고 해도 걱정할 필요 없다. 새로운 코드를 배정하면 되니까! 각 민족들은 그 민족의 언어와 주변 환경의 영향을 받아서 자신들 만의 문자 코드를 만들어 사용해 오고 있다. 영어나 중국어, 일본어라는 문자 코드를 한번 생각해보자. 그리고 그 코드의 표현 효율성에 대해서 비교하여 보자.

우리가 앞선 예에서도 보았듯이 코드를 만드는 것은 가능한 적은 재원으로 가능한 많고 다양한 정보를 표현하고자 하는 코드의 효율성이 중요하게 생각된다. 사실, 우리가 코딩하는 목적도 정보를 표현하는 것도 있지만 이미 표현된 정보도 더 효율적으로 표현하기 위한 목적도 있다.

효율적인 코드

우리는 일상생활에서도 코드를 만들어 내어 사용하기도 한다. 예를 들어 패스트푸드 음식점인 켄터키프라이드치킨을 각 단어의 앞 스펠링만 붙여서 KFC라고 하기도 하고, TV프로그램 '웃음을 찾는 사람들'도 마찬가지 방식으로 '웃찾사'라고 하기도 한다. 또 경찰에서도 코드를 많이 사용하고 있는데 예를 들면 '총기를 사용한 강도살인 사건'을 '코드234'라고 부를 수도 있는 것이다. 왜 그렇게 하는가? 더 효율적으로 사용하기 위해서이다. 같은 정보를 표현하는데 전자는 너무 길어서 매번 말할 때에도, 글을 쓸 때에도 코딩에 시간과 공간이 너무 많이 들기 때문에, 다른 정보와 겹치지 않는 범위내에서, 그리고 현재 사용되는 코드로 상호간 이해할 수 있는 범위내에서, 최대한 효율적인 (크기가 작은) 코드를 만들어 사용하고 있는 것이다. '웃찾사'와 같은 경우는 글자의 첫음을 모아서 줄인 경우이지만, 좀 다른 방법으로도 코딩의 효율을 높일 수가 있다.

동네 슈퍼마켓을 운영하는 아저씨는 매일 가게에 들여 놓을 물건을 전화로 주문을 한다. 예를 들어 생수 두 박스, 라면 한 박스, 백열전구 10개, 손톱깍이 1통, 머 그런 식이다. 매번 물건 이름을 이야기 하는 것이 번거스러우니까물건에 대한 약속된 코드값을 매겨서 사용한다. 그래서 '강릉 생수 700ml짜리 두 박스' 이렇게 이야기 하지 않고 '18766번 두 박스' 이렇게 이야기 한다. 훨씬 효율적이 되었다. 하지만 가만히 생각해 보면, 매일 주문하는 생수에도 5자리 코드를 쓰고, 한달에 한번 주문하는 손톱깍이에도 5자리 코드를 쓰는 것은 뭔가 비효율적일 것 같다. 따라서 예를 들어, 매일 주문하는 생수는 2자리 코드를, 아주 가끔 주문하는 물건에는 7자리 코드를 사용하여도 더 효율적이 될 수도 있을 것이다. 즉, 항상 사용되는 코드에는 짧은 것은 할당 하고, 가끔 사용되는 코드에는 긴 것을 할당해서 전체 정보 전달을 효율적으로 한다는 것도 효율적인 코드화의 한 방법이다. 몇가지의 사례를 보도록 하자.

'잠수복과 나비'는 1997년에 번역본으로 나온 책인데 2007년에 영화로도 나와서 (영화 제목은 '잠수종과 나비') 2008년 골든글로브 수상도 하였다. 프랑스의 잡지 편집장으로 일하던 저자는 젊은 나이에 갑자기 뇌졸중으로 쓰러지고 3주 후에 의식을 회복

해지만 그가 움직일 수 있는 것은 왼쪽 눈꺼풀 뿐이었다. 그가 15개월 동안 20만번 이상 눈을 깜빡여서 완성한 책이 바로 이 책이다. 다시 생각해보자. 보비가 의식을 회복 했을 때, 그는 모든 생각을 정상 처럼 할 수 있었다. 하지만 그가 움직일 수 있는 것은 왼쪽 눈꺼풀 뿐이었다. 언어도 사용하지 못하고, 문자도 사용하지 못하고, 손가락도 사용하지 못한다. 무엇을 사용하여 어떻게 내 머릿속의 생각을 '표현'하여 다른 사람에게 전달할 것인가. 어떻게 의사소통을 할 것인가. 그의 집필을 도와주는 사람이 매일 병실에 와서 스펠링을 a에서 부터 z까지 순서대로 하나씩 읽는다. 그러면 보비는 원하는 스펠이 나올 때 눈을 깜빡여서 표시한다. 그러면 그 스펠을 적고, 또 다시 보비에게 스펠을 하나씩 읽어 주는 식이다. 이것이 머릿속으로는 정상적으로 모든 것을 생각하고 있던 그가 바깥의 세상과 의사소통하기 위해 사용할 수 있었던 유일한 방법인 것이다. 그 과정을 코드의 효율적 관점에서 좀 더 다른 방법을 생각 해 볼 수 있다. 자주 사용되는 스펠을 먼저 불러 준다면, 시간과 노력을 많이 단축 시켜줄 수 있을 것이다. 프랑스어에서 각 스펠의 발생 빈도수를 보면 'E S A R I N T U L O M D P C F B V H G J Q Z Y X K W'의 순서라고 한다. 그러면 보비에게 이 순서대로 스펠을 불러 준다면 훨씬 효율적이 될 것이다. 인기 있는 미국의 퀴즈 프로그램인 'Wheel of Fortune'에서도 가려져 있는 속담을 알아 맞추기 위해 참가자 들이 순서대로 스펠을 하나씩 불러서 풀어 내는 프로그램인데, 참가자들이 제일 처음 부르는 스펠은 거의 대부분 'e'이다. 즉 영어에서 제일 많이 나오는 스펠을 차례대로 불러서 처음의 힌트를 잡아 내려고 하는 것이다.

비슷한 사례는 '모스부호(Morse Code)'에서도 찾아 볼 수 있다. 모스부호는 전화가 만들어지기도 전인 1800년대에 장거리 간의 문자 전송을 위하여 만들어진 '코드'이다. 1836년에 미국의 모스는 모스부호와 전신기(electrical telegraph)를 개발하였다. 전선으로 연결되 전신기의 키를 손가락으로 누르면 전류가 통하고 띄면 전류가 차단되는데, 키를 '띡" 짧게 누르거나, "띠이익-"하고 길게 누르거나 해서 그 조합으로 문자를 표시하였다. 그러면 전선의 다른 쪽에서는 그 "띡" 혹은 "띠이익"의 조합을 모스코드로 다시 재 해석해서 텍스트를 받는 식이었다. 1800년대 말에는 이탈리아에서 전파를 이용한 무선 전신기도 개발하여 장거리 통신 발전에 큰 바탕이 되었다. 자 그러면, 키를 한번 누르는 것은 두 가지 정보 (짧게 누르거나, 길게 누르거나)를 표현한다. 영어 스펠 26개와 숫자 10개를 합치면 36개 이므로, 표현 단위(즉 2가지 정보 표현)를 6개 나열하여 조합을 만들면 2x2x2x2x2x2 = 64 가지를 표현할 수 있으므로 (5개를 사용 하면 32가지가 되어 모자란다) 각 코드를 6개의 길이로 만들면 될 것이다. 하지만, 장거리의 수신자에게 적은 노력으로 적은 시간에 더 많은 텍스트를 보내야 하는 입장에서는 당연히 일융적인 코드가 아니라 더 효율적인 코드를 만들 수 있을 것이다. 그렇게 해서 모스는 영어에서 더 자주 쓰이는 철자에 더 작은 길이의 코드를, 덜 자주쓰이는 철자에 더 긴 코드를 배정하였다. 이 과정은 흔히 알고 있는 허프만 코딩 방법과 비슷하지만 허프만 코딩 알고리즘은 1950년대에 만들어 진 것이고 모스코드는 1800년대 초중반에 만들어 진 것이어서 개발이 상호 직접 연관된 것은 아닐 것이다. 기록에 의하면 모스는 영문자들의 일반적인 분포를 관찰하여 이 코드를 만든 것으로 되어 있다. 따라서 조금 불완전한 부분도 있다. 그리고 이 모스 코드는 문자를 이진코드와 비슷한 모습을 보이고 있는데, 아마도 1900년대의 디지털 이진 코드 개념에 큰 영향을 미쳤을 것으로 추측된다.

발생확률이 높은 것에 작은 코드를 부여하고, 그 반대의 것에 길이가 긴 코드를 부여하여 효율성은 높힌 것은 앞서 이야기 했던 모스 부호와 허프만코딩 알고리즘 이외에 또 다른 것과 연관되어 있는데 그것이 바로 '새논(Shannon)'의 정보이론(information theory)이다.

정보이론 (Information Theory): 이 정보의 가치는?

앞서 언급하였지만, 정보는 손으로 만져 질 수 있는 것도 아니어서 부피나 무게로 측정할 수 잇는 것도 아니고, 정보가 가지고 있는 '의미'는 사람마다 다르게 받아들여지는 것이어서, 정보의 가치는 '정량적으로 측정'할 수 있는 어렵다. 새논(Shannon)은 정보(information)을 정량적으로 측정가능하게 하는 이론을 제시하였다. 기본적인 아이디어를 간단히 설명하면 다음과 같다.

먼저 정보를 불확실성(Uncertainty)과 연결시켰다. 우리가 어떤 판단을 못하고 있다고 하자. 그 판단을 하기에 필요한 정보가 없기 때문이다. 그러면 그 상태에서의 불확실성(uncertainty)가 있을 것이다. 그 다음에 누군가 정보를 내게 주었고, 그 정보로 인하여 그 불확실성이 줄어 들었다. 그러면 그 줄어든 만큼의 불확실성 값을 그 정보의 가치라고 말할 수 있을 것이다. 즉, 어느 특정 정보의 가치는 그 정보로 인하여 줄어든 불확실성의 양이라고 정의할 수 있다. 그러면 그 불확실성(uncertainty)는 어떻게 정량적으로 측정하는가. 확률을 이용하여 측정할 수 있다. 하나의 예를 보자. 내가 동전을 위로 던져서 땅에 떨어졌을 때 앞면이 나올 것인가 뒷면이 나올 것인가. 불확실성이 매우 높다. 앞면일지 뒷면일지는 50%대 50%으로 같으므로 최고의 불확실성이다. 헌데, 누군가가 나에게 '앞면에 무거운 특수 자석을 붙여놓아서 뒷면이 위에 나올 확률이 높다'라는 정보를 주었다고 하자. 결국

앞면이 나올 확률 20%, 뒷면이 나올 확률이 80%하고 하면 불확실성은 그만큼 줄어 들었고, 만약에 각각 나올 확률이 0%와 100%라고 하면, 불확실성은 0이 된다. 여기서 그 정보의 가치는 그 정보가 없었을 때의 불확실성과 그 정보가 들어온 후의 불확실성의 차이만큼으로 측정하면 될 것이다.

새논은 불확실성을 확률값과 로그(logarithm with base 2)를 사용하여 표현하였다. Base가 2인 logarithm에서는 단위가 bit가 된다. 예를 들어 아까 양면이 나올 확률이 같은 동전의 경우에서, 가능한 output 심볼이 두 개이니까 (앞면, 뒷면) log_2(2) = 1 bit가 된다. 만약 동전에 무슨 조작을 해서 앞면만 100% 나올 것이라는 것을 안다면 가능한 output 심볼은 앞면 하나 이므로 log_2(1) = 0 bit. 즉 불확실성이 0비트이다. 두개 심볼의 확률이 같은 경우는 불확실성이 최대이고 1비트가 된다. 만약에 동전이 아니고 사면체 물건이라고 한다면 4가지의 symbols가 나오므로 log2(4)=log2(2^2)=2bits가 된다. 만약에 팔면체 주사위라고 한다면 log2(8)=3bits가 된다. 즉, 여러 개 더 많은 것들 중에서 골라야 하는 상황이니까 맞출 가능성이 더 낮아지고 불확실성이 더 높아지게 된다. 나올 수 있는 symbols의 수가 많아질수록 uncertainty는 증가한다. 이것은 현재 컴퓨터에서 모든 정보처리를 bit단위로 하는 것에 이론적 기반을 제공했다. (2가지 정보를 표현하려면 1bit가 필요하고, 8가지 정보를 표현하게 하려면 3bits가 필요하다.)

자 이번에는 각각의 심볼이 나올 수 있는 확률이 각각 다른 경우를 생각해보자. 각 심볼에 대한 확률을 사용해야 하는데, 먼저 아까 처럼 모든 심볼의 확률이 똑 같은 경우에서부터 시작해보자.
이 경우에 Uncertainty 는 log2(M) 그리고 M은 심볼의 개수 였다.
log2(M) = −log2(1/M)이며, 1/M은 확률p와 같으므로 −log2(p)라고 쓸 수 있다.
동전의 경우에는 M=2이므로 log2(2)=1이며, 또 각 심볼이 나타날 확률은 50%, 즉 1/2이므로 −log(1/2)=1 이 된다.
이것을 좀 더 일반적으로 정리해보자.
M개의 심볼이 있고, 각 각각의 index를 i 라고 하자. 그리고 그 i번째 symbol이 나타날 확률을 p_i라고 하자.
확률의 합은 1이 되어야 하므로 SUM_{i=1}^M P_i = 1이 되어야 한다.
i번째 symbol이 나타났을 때 우리가 겪게 되는 놀라움(surprise)를 “ surprisal” 이라고 부르고 그것을 다음과 같이 정의하자.
u_i = −log2(p_i)
만약 나타날 확률이 적은 놈이 나타난다면 우리의 “놀라움”은 매우 클 것이고, 나타날 확률이 매우 높은 놈이 나타나면 우리는 별로 놀라지 않을 것이다. 즉, p_i가 0에 가깝다면 i번째 symbol을 보게되면 very surprised 할 것이어서 u_i=infinity 할 것이고, 만약 p_i=1 이라면 “전혀” 놀라운 일이 아니라서 u_i=0이 될 것이다.

이제 uncertainty를 생각해보자. M개의 각 symbols의 u_i (즉, 개별 uncertainty)의 평균을 계산함으로써 device(즉 동전, 주사위 등)의 “uncertainty”을 계산한다. 평균은 어떻게 계산할 것인가?
평균은 즉 average surprisal (H) 은 H = SUM p _i * u_i = − SUM p_i * log2(p_i)가 되고
정리하면 H = SUM −p_i*log2(p_i) bits. 이 H를 정보의 엔트로피라고 한다.
우리가 어떤 심볼의 발생확률을 모를 때에는 발생 횟수를 카운트 해서 확률로 만들어 사용할 수가 있다. 예를 들어 A,B,C,D의 네개의 심볼이 있는데 전체 100개의 발생중에서 각각 60, 20, 10, 10 번 발생 했다고 하면, 각각의 발생확률은 6/10, 2/10, 1/10, 1/10이 되고 각각의 u값을 구하고 그것을 각각의 확률에 비례해서 평균을 내는 것이다. 만약 발생확률이 모두 25/100=1/4로 똑같다고 한다면, −1/4log2(1/4) −1/4log2(1/4) −1/4log2(1/4) −1/4log2(1/4) = −log2(1/4) = log2(4) = 2bits가 된다. 이 경우는 log2(M)=log2(4)=2bits와 똑 같은 값을 낸다.

이해를 돕기 위해 간단한 예를 보도록 하자. DNA의 어느 position에 symbol A C G T 중의 하나가 발생을 한다. 어떤 symbol이 발생할 줄 모르면 우리는 equally likely하다고 가정을 하고, 그렇다면 이 position이 가지는 uncertainty는 log2(4)=2bits가 된다. Uncertainty를 줄이고자 몇 가지의 sequence를 관찰을 했다. 그랬더니 그 position에서 최근 발생했던 심볼은 ACATGAAC 들이었다. 그러면 이 position이 가지는 uncertainty, 혹은 entropy는 얼마나 될까.
관찰에서 각 심볼의 발생횟수를 가지고 발생확률을 계산한다. p_A = 4/8 = 1/2, p_C = 2/8 = 1/4, p_G = 1/8, p_T = 1/8이다.
H = −1/2log2(1/2)−1/4log2(1/4)−1/8log2(1/8)−1/8log2(1/8)) 이 되고 그러면 H = 1/2+2/4+3/8+3/8 = 14/8 = 1.75bits 가 된다.
즉 entropy가 2bit에서 1.75bit로 줄었다. 우리가 가진 관찰 정보 때문에 entropy가 그만큼 줄었으며, 그것이 그 정보의 가치이

다.

wheel of fortune 퀴즈쑈의 예를 다시 생각해보자. 간단한 예를 만들어 보면, ? ? ? ? 의 네 글자의 단어를 맞추는 것이라고 하자. 참가자는 글자를 하나씩 불러가면서 단어를 맞혀야 한다. 최고의 entropy이다. 철자가 26개이고 4개가 연속해 있으니까 26^4 = 456,976개 중의 하나일 것이다. 그 모든 조합이 발생할 확률이 모두 같다고 가정한다면 엔트로피는 log2(456976) = 18.***bits가 될 것이다. 하지만 우리는 앞서 보았던 각 철자의 발생확률을 알고 있다. 이것을 이용하면 엔트로피를 많이 낮추는 방향으로 퀴즈를 풀어 나갈 수가 있는 것이다.

모든 종류의 데이터들은 하나의 통일된 형태로 encoding되어야 효율적일 것이다. 이 통일된 형태를 우리는 bit pattern이라고 부른다.
“bit” (binary digit) 은 데이터의 가장 작은 단위를 말한다. 한 bit는 0또는 1을 표시할 수 있다. 우리의 집에 있는 전기 스위치는 전기를 켜고 끄는 역할만 담당한다. 스위치는 1bit의 정보를 저장할 수 있는 기구라고 말할 수 있다.
2bit를 가지고는 몇 가지 정보를 저장할 수 있을까. 2^2=4가지, 즉 00, 01, 10, 11 의 4가지 정보를 저장할 수 있을 것이다. 8bit는 2^8=256가지 정보를 저장 할 수 있다.
만약에 우리가 A, B, C, D, E 라는 정보를 저장할 기구를 만들려고 한다면 몇bit만 있으면 될까. 2bit이면 4가지만 저장 할 수 있으므로 부족하다. 3bit이면 8가지를 저장할 수 있으므로 문제를 해결할 수 있다. 예를 든다면 000에는 A를 나타나게 하고, 001에는 B를 나타나게 하고. 물론 3개는 남게 될 것이다.

우리 수업에 70명이 등록을 하였는데, 70명의 이름을 저장하려고 하면 몇 bit가 필요할까. Log_2^70을 하게 되면 7bit가 필요할 것이다. 7bit만 있으면 70명의 이름을 각각 구분할 수 있게 되고도 남는다.
우리 인간의 뇌에는 10^11개의 neuron이 있다고 말한다. 각각의 neuron은 한 bit정보를 처리한다고 가정을 한다. 이론적으로 말하자면 2^(10^11)개 만큼의 정보를 처리할 수 있다는 것이다.
컴퓨터에서는 하나의 정보를 처리하는 단위로 byte를 표시한다. 1byte는 8bit를 말한다.

데이터 인코딩

컴퓨터에서는 그 회로에서 전기흐름의 연결/비연결의 형태로 데이터를 처리한다. 즉 연결(1)과 비연결(0), 즉 1과 0이라는 두 가지 표현을 사용한다. 따라서 컴퓨터에서 데이터를 사용하게 하기 위하여서는 1과 0이라는 이진수로 변환, 즉 인코딩을 해주어야 한다. 물론 이러한 인코딩을 우리 일반 사람이 매번 해줘야 되는 것은 아니다. 예를 들어 우리가 디지털 카메라로 사진을 찍는다면, 그 사진은 디지털 인코딩 되어 저장이 되는데, 그 인코딩은 컴퓨터과학자들이 이미 만들어 놓은 인코딩 방법에 의하여 자동으로 처리된다. 즉 1과 0 이라는 이진수 형태로 저장되고 처리되는 것인데, 문제는 그 인코딩 방법은 여러가지고 있으며 또 새롭게 계속 개발되고 있다는 것이다. 예를 들어 우리는 그 디지털 카메라에서 jpg, bmp, png 등의 인코딩 방법을 선택할 수 있으며 그 품질정도도 선택할 수 있다. 그것에 따라서 다르게 인코딩되어 저장되는 것이다. 따라서 우리는 컴퓨터에서 사용되는 다양한 인코딩 방법에 대하여 알 필요가 있으며, 또한 그러한 인코딩 방법을 이해 하는 과정에서 정보라는 것이 어떻게 디지털 코딩이 되는지에 대한 근본적인 이해를 할 수 있게 될 것이다.

여기서는 문자, 이미지, 소리 그리고 숫자를 어떻게 이진 데이터로 변환하는지를 보도록 하자. 일단 디지털 코딩, 이진 코딩이 되면 컴퓨터라는 기계에서 알고리즘에 의해 계산 처리 될 수 있게 된다. 아래한글이나 MS word같은 워드프로세서 프로그램은 문자 데이터를 만들고, 없애고, 옮기고, 모양을 바꾸고 하는 등의 일들을 처리한다. 이미지 처리 프로그램은 이미지 데이터를 확대/축소도 시키고, 회전도 시키며, 각종 효과도 처리할 수 있다. 음악이나 목소리도 디지털 데이터로 변환되어 처리되고, 숫자는 각종 과학 계산 소프트웨어에서 효율적으로 처리 될 수 있는 형태로 인코딩 된다.

문자(text)데이터 인코딩

어떤 언어에서 text라는 것은 생각을 표현하기 위한 기호들의 순서적 나열 (a sequence of symbols)이다. 영어라는 언어에서 보면 사용되는 기호들이 대문자 26개(A, B, C, .., Z), 소문자 26개(a, b, c, .., z), 그리고 숫자기호 9개 (0, 1, ..,9), 그리고 기타 기호

들(.,!,? 등)으로 이루어져 있다. 이제 각 기호들을 이진데이터로 어떻게 인코딩을 해야 하는지를 생각해 보자. 우리가 앞서서 보았던 bit pattern을 다시 생각해보자. 그러면 문제는 하나의 기호를 표현하기 위해 면 bits가 필요할 것인가일 것이다. 그리고 그것은 전체 기호의 개수에 의해서 결정된다. 만약 4가지 다른 기호를 표현하기 위해 2bit (log2(4)=2)를 사용하면 (즉 00, 01, 10, 11) 되는것 처럼, 비트의 수는 전체 기호가 몇 개몇개 따라서 달라지게 된다. 영어의 예를 보면 대소문자와 숫자, 그리고 기호 합쳐서 대략 70-80개 정도 될 것으로 추정할 수 있는데, 그렇다면 몇 bit를 사용하면 그 기호들을 모두 표현할 수 있을까. 80개로 가정하고 로그를 취하면 log2(80)=6.32 가 되므로 최소한 7bit면 다 표현을 할 수 있겠다. 왜냐하면 6bit만 사용하면 64개만 표현할 수 있어서 모자라지만 7bits를 표현하면 128개를 표현하므로 충분히 여유 있게 사용할 수가 있게 된다. (한글이나 한자는 어떨까?)

그러면 그 다음엔, 어느 기호를 어느 bit pattern에 배정을 할 것인가에 대한 문제이다. 이것에 대해서는 '약속된 표'를 만들어서 사용하면 된다. 즉, 이 기호는 이 bit pattern으로 하자라는 약속 표를 만들어서 'E'가 1000101이다 라는 것을 사람들이 알고 사용하게 하면 되지 않는가. 이 표를 우리는 character set이라고 부른다. 영어에서는 아까 생각해 본 것처럼 7bit만 있으면 모든 대소문자, 숫자, 기호를 다 표현할 수 있을 것이며 그것에 대하여 각 기호에 bit pattern을 배정한 character set중에 가장 많이 알려진 것이 ASCII character set 이다

ASCII code는 각 symbol에 대하여 1byte, 즉 8bit를 사용한다. 즉 256가지의 정보를 표현할 수 있다. 8bit중 첫 번째(제일 왼쪽) bit는 check bit로 사용하고, 나머지 7bit를 사용하여 128가지의 정보를 표현하는데, 영어권에서는 이것으로 모든 문자와 심볼을 나타낼 수 있다. 아래의 표를 보면 각 기호들이 0부터 127까지 모두 128개의 숫자에 배정되어 있다. 대문자 'E'를 아래표에서 보면 69번째 기호이며 69는 이진으로 1000101이 된다.

컴퓨터를 처음 활용했었던 영미권 국가에서는 이 문자코드표로 문제가 없었지만, 이후 비 영어권 국가에서도 컴퓨터를 사용하게 되었고 따라서 그들의 문자를 표현할 수 있는 문자코드표의 필요성이 생겼다. 그래서 나온 것이 Unicode이다. Unicode는 2byte, 즉 16bit를 사용한다. 즉 2^16=65,536가지의 문자를 표현할 수 있다. 이 Unicode에는 한글을 포함하여 전세계 모든 문자들을 이 하나의 character set에 다 포함시킬 수 가 있다. 물론, 한자는 약식한자만 포함된다. 앞서서 보았지만 로마자를 사용하는 미주와 유럽국가들에서는 문자의 수가 그리 많이 않지만 한글과 한자, 일본어와 같은 문자가 많은 비중을 차지하고 있다. 지금은 세계 모든 문자들과 기호들을 모두 포함하여 사용하고 있다.

우리가 사용하는 text의 각 문자 (spell, digit, symbol)들이 어떻게 bit pattern으로 바꾸는지에 대하여 살펴 보았다. 이제는 어떻게 "효율적"으로 저장할 것인가에 대하여 생각하여 보자.

"Korea"는 5개의 문자로 이루어져 있다. Unicode를 사용한다고 하면 우리는 5symbols*16bits = 80bit의 메모리가 필요할 것이다. 만약에 엄청나게 많은 text정보를 가지고 있는 문서가 있다고 할 때, 그것을 bit로 바꾼다고 할 때, 메모리의 낭비가 최소한으로 할 수 있는 방법은 없을까.

문자데이터의 압축

코딩의 효율성은 '얼마나 적은 bit를 가지고 얼마나 많은 정보를 표현할 것인가'라고 할 수 있다. 문자들로 이루어진 문서를 코딩할 때에 그 문서에 대한 약간의 힌트와 지식을 이용해서 전체 bit수를 줄일 수 있다면 많은 도움이 될 것이다. 이렇게 줄이는 것을 일반적으로 압축(compression)이라고 말하며, 여기서는 다음과 같은 기본적인 압축 방법 몇가지를 영문자를 가지고 설명하도록 하겠다.

- Keyword encoding (키워드 인코딩)
- Run-length encoding (반복길이 인코딩)
- Huffman encoding (허프만 인코딩)

Keyword encoding은 문서에서 매우 자주 사용되는 단어에 대하여 특별히 짧은 코드를 부여하여 전체 크기를 줄이겠다는 것이다. 에를 들어 영어문서에서 the, and, which, that, what 등과 같은 단어는 아주 자주 나오는 단어들이며, 이 단어들의 코딩 bits 수만 조금만 줄여도 전체적으로는 도움이 될 것이라는 생각에서 나온 것이다. 예를 들어 자주 등장하는 키워드들에 대하여 다음과 같은 짧은 기호로 대체한다고 하자.

즉 2-5자의 문자로 이루어진 단어가 1개의 문자로 줄어 들었다. 여기에 따라서 다음의 문장을 인코딩한다고 하면 다음가 같이 변환될 것이다.

원래 문장에서는 349개의 문자가 사용되었으나 압축된 것에서는 314개의 문자가 사용되어 35개의 문자가 줄어 들었으며, 따라서 314/349=0.9의 압축률을 보인다.

이 방법은 여러 가지 문제점들이 있는데, 예를 들어 '$'이 실제 dollar sign인지 아니면 that의 대체기호인지를 분간할 수가 없다. 따라서 사용되는 대체기호는 제한 될 수 밖에 없다. 또 다른 문제는, 아마도 가장 많이 사용되는 단어 중의 하나인 a 나 I 같은 것은 어차피 한 개의 기호이므로 압축효과가 없다. 하지만 만약 어떤 특정 분야의 문서에서 특정 단어가 많이 사용된다면 그것을 키워드로 사용함으로써 많은 도움을 받을 수가 있다. 예를 들어 동물에 대한 문서에서 'Siberian Husky'라는 단어가 많이 나온다면, 이런 특정 키워드 단어를 대체기호로 사용하면 압축을 늘릴 수 있을 것이다. 또 다른 것은 만약 특정한 단어가 아니라 단어의 일부를 압축한다면 더 압축률을 늘릴 수 있을 것이다. 예를 들어 ex-, -ing, -tion 등은 많은 단어에서 사용되는 부분이므로 대체기호로 바꿀수도 있을 것이다. 그럼에도 불구하고 키워드 인코딩은 압축되는 양은 그리 크지 않다.

Run-length (반복길이) 인코딩은 하나의 문자가 계속 반복해서 나열되는 경우에 사용된다. 실제 일반 문장에서는 거의 나타나지 않는 경우이지만 DNA염기서열 데이터 같은 경우에 많이 발생하기도 한다. 일반적으로 반복기호열을 반복표시기호-반복기로-반복횟수 이렇게 3문자로 대체하는 방법이다. 예를 들면 kkkkkkkkk 을 *k9 로 표시하는 것이다. 여기서 *를 표시기호 라고 가정한다. 반복횟수는 1자리 숫자만 사용한다고 가정한다. 예를 들어 인코딩된 문장이라고 한다면 이 경우에 원래 51개의 문자가 35개로 줄었으므로 압축률은 35/51=0.68이 된다. 여기서 ccc와 eee는 인코딩되어 있지 않은데 인코딩 해봐야 똑같이 3개의 문자이어서 압축되지 않기 때문이다. 그리고 우리의 가정에서 반복횟수를 1자리 숫자로 표시한다고 했는데 그러면 9번까지의 반복만 표시할 수 있을 것 같아 보이지만, 실제로는 각 문자가 이진데이터로 바뀌어 8비트를 사용한다고 하면, 반복횟수는 어차피 8바트로 사용하기 때문에 0-255까지, 혹은 4개 이상 반복만 인코딩 하므로 4-259번까지의 반복횟수를 표수할 수 있다.

허프만 코딩은 David Huffman박사의 이름을 딴 것인데, 거의 잘 사용되지 않는 문자 'x'와 가장 많이 나오는 'e'에 같은 수의 bit 을 사용할 필요가 있는가라는 생각에서 나온 방법이다. 앞서서 우리가 공부한 모스부호 코드를 보면 문자의 일반적인 발생빈도에 따라서 다른 길이의 코드를 부여한 것을 보았는데 그것과 같은 것이라고 할 수 있다. 다만 허프만코딩은 어느 특정 문서에 대하여 그 문서에 나오는 각 문자의 빈도수를 계산하여 그것에 따라서 코드표를 만들어 주는 방법을 보여주고 있다. 많이 발생하는 문자에는 짧은 코드를 발생 빈도수가 낮은 문자에는 긴 코드를 부여하여 전체 코딩 크기를 줄이려고 하는 방법이다.

먼저 인코딩과 디코딩에 대하여 간단히 살펴보도록 한 후에 구체적인 방법에 대하여 보도록 하자. 다음의 몇 개 문자에 대해 허프만 코드가 다음과 같이 정해졌다고 하자.
자주 발생하는 A, E에는 길이 2의, 자주 발생하지 않는 D에는 길이 4의 코드가 배정되었다. 예를 들어 DOORBELL이라는 문장은 다음과 같이 인코딩 된다.
만약 각 문자당 8비트씩 공일하게 사용하는 ASCII코드를 사용한다고 한다면 DOORBELL이 모두 8자이므로 8x8=64bits가 필요하지만, 허프만코딩을 사용한다면 모두 25bits가 사용되어, 25/64=0.39의 압축률을 보이게 된다.
자 그러면 반대로 위의 25bits의 이진코드를 봤을 때 어떻게 월애 문자로 디코딩 할 것인가. 만약 아스키코드를 썼을 때에는 8bits씩 잘라서 각각을 디코딩하면 되었지만, 허프만 인코딩에서 처럼 각 문자의 bits수가 다를 경우에는 어디서 잘라서 디코딩해야 할 것인지 헷갈리는 것이다. 예를 들어 위의 인코딩된 것을 보면 앞에서부터 10을 가지고 해야할지, 101을 가지고 해야할지, 혹은 1011을 해야할지 혼란스럽게 된다. 하지만 다행히 허프만코드표를 보면 10, 혹은 101에 해당하는 문자가 없으므로 1011에 해당하는 D라고 디코드하면 된다. 이렇게 어떤 문자코드는 다른 문자코드의 prefix(앞부분의 일부)가 되면 절대로 안된다. 모스부호도 이 조건을 만족하고 있으며, 이것이 허프만인코딩 방법이 만족시키는 기본 조건이기도 하다. 자 그럼 어떻게 허프만코드를 만들어 내는가 보기로 하자.

예를 들어, 우리가 코딩하려고 하는 문서는 다음과 같은 7개의 문자들로만 이루어져 있다고 하자.
K, O, R, E, A, space, newline
먼저 ASCII코드를 사용하면 각 문자달 8bits가 필요하다. 만약 이 문자들만 가지고 별도의 코드표를 만든다면, 모두 7개의 문자이므로 log2(7) 이므로 각 문자당 3bits면 충분하다. 각 문자에 3bit의 pattern을 부여하고 문서에서 각 문자의 발생빈도(frequency)를 조사하여 보았다.

문자	코드	빈도(frequency)		전체 사용 비트 수
K	000	100	300	
O	001	140	420	
R	010	110	330	
E	011	50	150	
A	100	70	210	
Space	101	130	390	
Newline		110	15	45
total		615개 문자		1845 bits

이 문서는 각 문자당 3비트를 사용하였을 때 전체 1,845비트를 사용하였다. 만약 각 문자당 8bits를 사용하는 ASCII코드를 사용했다면 4,920bits을 사용하게 될 것이며, 유니코드를 사용하면 9,840bits를 사용하는 것이다.
이 3 bit-pattern은 다음 그림과 같은 이진 트리로 표현될 수 있다.

이 트리에서 각 문자의 코드는 다음과 같이 찾아진다. 트리의 root node(제일 꼭대기 노드)에서부터 시작하여, 왼쪽 가지로 가면 0, 오른쪽 가지로 가면 1이 붙여진다. 예를 들어 R이라는 문자는 root부터 시작하여 왼쪽으로 한번, 오른쪽으로 한번, 다시 왼쪽으로 한번 가서 만날 수 있으므로 R의 코드는 "010"이 된다. (이러한 데이터 구조를 trie라고 부른다). 이 트리는 우리가 위의 표에서 사용한 코드를 보여주고 있다.
만약 이 트리를 다음과 같이 개선시키면 좀 더 효율적이 될 수 있다.

제일 오른쪽 노드에는 chile node가 하나 만 있었기 때문에 위로 올려도 무관하다. 그래서 nl은 한 칸 위로 올라갔고, 따라서 nl의 코드도 "11"이 되어 2bit만 사용하게 된다. 따라서 이 문서의 전체 사용 bit수도 15bit만큼 줄어들고 전체 사용 비트수는 1830 이 된다.

이러한 트리에서 문자가 제일 아래 끝에만 위치하게 된다면, 이 트리에서 만들어지는 bit sequence는 분명하게 명확하게 해석될 수 있다. 즉, 어느 한 문자의 코드는 다른 문자의 코드의 prefix가 되지 않는다는 것이 보장된다. 각 문자의 코드 길이가 다르다 하더라도 이것은 보장된다. 이러한 인코딩을 prefix code라고 부른다.
문제는 어떻게 각 문자의 발생빈도를 고려한 효율적인 코딩 트리를 어떻게 만들 것인가이다. Huffman이 1951년에 만든 알고리즘을 소개한다. 이 코딩 시스템을 보통 Huffman code라고 부른다.

이것을 다시 정리하여 보면

문자	코드	빈도(frequency)	전체 사용 비트 수
K	000	100	300
O	01	140	280
R	001	110	330
E	1110	50	200
A	110	70	210
Space	10	130	260
Newline	1111	15	60
total		615개의 문자	1640bits

즉, 전체 1640비트를 사용하게 되어 3비트 균일 인코딩에 비하여 88.8%의 압축률을 보이며, 8비트 아스키 인코딩에 비하면 33.3%의 굉장한 압축률을 보이게 된다.

유니코드는 원래 16bits를 사용하지만 (utf-16) 영어권 사람들은 원래 8bits만 가지고도 문제가 없었기 때문에 문자당 16bits를 사용한다는 것은 효율성과 압축관점에서 보면 손해가 된다. 그래서 나온 것이 utf-8이며, 8bits만 사용하여 영문자를 처리하고 그 외의 것은 16bits, 24bits 이렇게 늘려가면서 사용하도록 되어 있다. 미국에서는 8비트만 사용하고, 유럽 변형 알파벳은 16비트를 사용하고, 한자/한글 등의 문자는 24비트로 늘어나게 된다. 사실 동양문자를 쓰는 사람은 utf-16을 사용하는 것이 더 유리하다. 하지만 미국처럼 영문자만 사용하는 경우엔 사용하지 않는 동양 문자를 제거하여 파일크기를 2배로 압축할 수 있게 된다.

이미지(Image) 인코딩(encoding)

자 이번에는 이미지를 어떻게 디지털화를 하여 컴퓨터에 저장 및 처리를 하게 할 것인가이다. 다시 이야기를 하면 이미지를 어떻게 1과 0으로 이루어진 나열로 변환할 것인가이다. 그리고 그 1과 0의 나열은 다시 원래 이미지로 다시 만들어져야 한다. 수수게끼 퀴즈 같기도 하고 암호를 만드는 작업 같기도 하지 않은가?

먼저 간단한 것부터 이야기 해나가도록 하자. 이미지는 먼저 가로 세로 격자로 나누어 잘게 쪼갠다고 생각해보자. 그러면 이미지는 잘게 쪼개어진 칸들의 행렬(matrix)처럼 될 것이다. 그 각각의 칸을 픽셀(pixel; picture element)라고 부른다. 아주 잘게 쪼개기 때문에 각 같은 실제로는 하나의 점 같은 크기이다. 그 픽셀의 개수를 그 이미지의 해상도(resolution)라고 부른다 만약에 한 이미지를 1000개의 픽셀로 쪼갤수도 있고 10000개의 픽셀로도 나눌 수도 있는데 후자가 물론 훨씬 좋은 해상도를 보이겠지만 그만큼 메모리를 많이 차지하게 될 것이다.

이미지를 펙셀들로 나눈 다음에 각각의 픽셀을 bit pattern으로 변경시킨다. 자 이제부터는 하나의 픽셀 가지고 bit pattern을 생각해 보도록 하자. 이미지는 black-and-white일 수도 있고, grey-scale일 수도 있고, 낮은 품질의 컬러일 수도 있고, 높은 품질의 컬러 일 수도 있다. 그 이미지가 무엇이냐에 따라서 그리고 어떤 품질을 원하느냐에 따라서 각각 여러가지 인코딩 방법이 사용될 수 있다. 먼저 아주 간단한 이미지를 생각해 보도록 하자.

먼저 흑백만 표시하는 (a)의 이미지가 있다고 하자. 이것을 (b)처럼 6x6 개의 픽셀로 나눈다고 하면 각 픽셀을 흑 또는 블랙만 표시하면 된다. 각 픽셀마다 흑 또는 백, 이렇게 두 가지만 표시하면 되므로 각 픽셀당 1bit만 있으면 된다. 그래서 이미지가 그 픽셀을 차지하면 1이라고 표시하고 그렇지 않으면 0이라고 인코딩하기로 하자. 그러면 우리는 (c)처럼 인코딩된 이미지를 보게 되는 것이고 이것의 bit pttern의 행렬은 (d)처럼 표현된다. 이 행렬 표현은 한줄로 나란히 배열 시켜도 된다. 이렇게 함으로써 (a)의 이미지는 (d)의 이진표현으로 인코딩될 수 있는 것이다. 디코딩은 반대로 하면 된다. 이렇게 디지털화를 하는 과정에서 아날로그 정보의 일부는 손실 되기도 한다. 따라서 아직도 고품질의 사진 촬영 작업은 아날로드로 하기도 하지만 최근에는 디지털 해상도가 워낙 높게 표현되는 기술이 발달해서 (즉 픽셀의 크가가 워낙 커지게 할 수 있어서) 육안으로는 구별하기 힘든 정도가 되었다.

사실 흑과 백, 이렇게 두가지로만 이루어진 이미지는 거의 없다. 그 다음으로 생각해 볼 수 있는 것이 컬러정보는 없지만 명도 정보는 다양하게 표현되어 있는 grey-scale 이미지를 생각해보자. 각 픽셀에 표시되는 정보는 흑과 백 뿐만 아니라 아래 그림과 같이 다양한 명암을 가진 회색계열 색을 표현할 수 있다.

자 그러면 각 픽셀당, 얼마나 많은 명도 정보를 표현할 것인가? 일반적으로 각 픽셀당 1byte, 즉 8bits를 배정한다면 0부터 255까지 모두 256가지의 명도 정보를 표현할 수 있을 것이다. 0은 완전 black, 그리고 255는 완전 white로 지정하여 encoding 할 수 있다. 그리고 명도의 차이에 따라 256가지 다른 코딩 값을 지정하면 될 것이다. 이렇게 되면 black-and-white의 이미지보다 8배나 많은 bits수가 필요하게 될 것이다. 6x6 픽셀의 이미지의 경우 흑백 표현은 6x6x1bits=36bits만 필요하지만, grey-scale의 경우에는 6x6x8bits= 288bits가 필요하게 된다. 아래 이미지는 540x720 픽셀을 가지고 있으며, 각 픽셀이 8bits grey-scale일 때와 1bit이 black-and-white일때를 비교해보고 있다. 오른쪽의 일부 확대 이미지를 보면 각 픽셀이 사각형 모양으로 보여지는 것을 볼 수가 있으며, 다양한 greyscale과 black and white로만 표시된 픽셀을 확인할 수 있다

색상의 표현

색상(Color)은 우리 눈의 망막(retina)에 도달하는 빛의 다양한 주파수에 대해 우리가 인식하는 것이다. 인간의 망막에는 빛의 주파수에 대해 다르게 반응하는 깔데기 모양의 빛 감지 세포가 세 가지가 있다. 각각은 빨강, 초록, 그리고 파랑의 색상에 해당하는 빛 주파수에 대하여 반응한다고 한다. 인간의 눈이 인식하는 모든 다른 색상은 이들 세 가지 색상의 강약 조합에 의해 만들어 질 수 있다. 따라서 이러한 망막의 기능을 흉내내어 이미지의 색상은 일반적으로 컴퓨터에서는 RGB(Red-Green-Blue)값으로 표현하는데, 실제로는 세개의 값인데 각각 red, green, blue의 값의 정도를 표시하고 있다. 각 값은 8비트를 사용하여 256가지의 scale을 가지는데 0은 그 색상이 아예 없는 것, 255는 그 색상이 완전한 것으로 표시한다. 예를 들어 RGB값이 (255,0,0)이면 완전한 빨간색이며, (255,255,0)이면 빨강과 초록을 극대화 시키고, 파랑을 없애버리게 되어, 노란색깔이 표현된다. 아래의 표는 몇가지 색상에 대한 RGB값을 보여주고 있다.

우리가 흔히 사용하는 많은 편집 소프트웨어에서도 사용자들이 RGB값을 이용하여 색상을 선택하도록 하고 있다. 아래의 그림은 MS 파워포인트 소프트웨어에서 RGB 값이 (79,129,189)인 색상을 보여주고 있다.

색상의 깊이를 나타내는 데이터의 양을 흔히 color depth, 즉 색상깊이라고 한다. 우리가 사용하는 TrueColor는 24bits의 color depth를 사용한다. 즉 세가지 기본 색상에 대하여 8bit씩 사용하여 하나의 픽셀에 대하여 24비트의 bit pattern을 사용한다.

RGB는 세가지 색상에 대해 각기 조합으로 표현되므로 일반적으로 3차원 공간으로 이야기 되기도 한다. 3차원 색상 공간의 각 꼭지점은 아래의 그림처럼 보여진다.

이 3차원 공간안에 Red, Green, Blue가 조합되어 만들어지게 되는 색상은 모두 256x256x256 = 16,777,216 가지인데, 우리의 눈이 구별할 수 있는 색깔은 그 중에 몇가지나 될까?

사실, 24bits을 사용하는 TrueColor는 인간이 구별할 수 있는 색상의 수보다 훨씬 많은 색상을 제공한다. 따라서, 이미지 파일 크기를 줄이기 위하여 indexed color (인덱스 색상)을 사용하기도 한다. 어떤 웹브라우저와 같은 응용프로그램에서는 모든 색상을 다 사용하지 않고 제한된 개수의 색상만을 가지고 팔레트를 만들어 사용자가 실제 색상과 가장 비슷한 색상을 팔레트에서 골라서 사용하도록 하기도 한다. 아래는 팔레트의 예를 보여주고 있다.

RGB색상 공간이 앞서의 그림과 같이 3차원의 큐브형태로 표현되었지만, 원통형 공간으로 표현되기도 하는데 대표적인 것이 HSL(Hue, Saturation, Lightness)과 HSV(Hue, Saturation, Value)이다. RGB 값을 큐브 공간에서 표현 하는 것 보다 더 직관적이고 인지적이라고 하며 많은 이미지 처리 프로그램에서 함께 사용되고 있다.

자 지금까지는 이미지가 어떻게 이진코딩이 되는지에 대한 기본적인 방법을 살펴보았다. 하지만 이미지의 인코딩에는 여전히 문제가 있는데 그것은 바로 인코딩된 파일의 크기 문제이다. 예를 들어, 24bits TrueColor사용한다면 540x720 해상도의 이미지는 각 픽셀당 24bits, 즉 3byte의 bit pattern을 사용하므로 파일크기가 540x720x3 = 1,166,400바이트가 된다. 즉 약 1.16MB (메가바이트)크기가 된다. 17인치 컴퓨터 모니터의 최대 해상도(resolution)로 본다면 1280*1024*24bits / 8 = 3,932,160 바이트 = 약 4MB. 엄청난 메모리를 차지하게 된다. 따라서 더 효율적인 인코딩 방법, 즉 적은 코드로 더 많은 정보를 인코딩 할 수 있는 압축 방법을 생각하게 된다.

효율적인 이미지 압축 코딩

우리가 각 픽셀에 대하여 똑 같은 크기의 bit pattern을 사용하는 것을 우리는 bitmap 비트맵이라고 부른다. 이미지 파일의 확장자가 .bmp 라고 되어 있는 것이 바로 비트맵 파일이라는 것을 의미한다. 일반적으로 픽셀당 24bits TrueColor 비트맵을 사용하는데, 앞서 문자 압축 기법에서 배웠던 반복길이(run-length)기법이 적용되기도 하고, 혹은 사용되는 색상의 종류가 많이 않다면 색상깊이(color depth)를 24bits보다 적게 사용하기 도 한다.

GIF(Graphics Interchange Format)형식은 1987년에 만들어 진 것인데 파일크기를 줄이기 위하여 색인 색상(indexed color)을 사용하여 256개의 색상만 사용할 수 있다. 256가지의 색상만 표현하게 하려면 픽셀당 8bits만 사용하면 되므로 픽셀당 24bits를 사용하는 비트맵보다 1/3을 줄일 수가 있다. 이러한 이유로 GIF는 라인아트(line art)같은 그래픽이나 혹은 사용되는 색상의 수가 많지 않은 사진의 경우에 주로 사용된다. 무료로 사용되던 GIF압축에 사용되는 LZW 압축알고리즘이 1995년부터 특허가 적용되기 시작하면서 사람들은 그 대안으로 무료압축형식인 PNG (Portable Network Graphics) 형식을 만들어 사용하고 있는데 GIF에 비하여 압축률도 좋고 TrueColor 색상을 사용한다.

가장 많이 사용되는 압축 이미지 형식인 JPEG은 사람 눈의 특성을 반영하여 만들었다고 한다. 쉽게 설명한다면 비슷한 색상끼리 모여 있는 영역을 평균화 시키는 방법이다. 여러 수학적 변환 모델을 사용하며 계속 발전해 나가고 있다. JPG는 주로 컬러사진에 많이 사용되며 문자, 만화 혹은 흑백 라인아트 같은 것에는 그 효율성이 다른 것에 비하여 떨어진다. 픽셀영역을 평균화 시켜서 압축하기 때문에 실제 원 정보가 손실된다. 따라서 JPG를 손실압축(lossy complession)이라고 부르기도 한다. 일반적으로 bmp에 비하여 10:1정도의 압축률을 보이는데 압축률은 사용자가 파일의 크기와 이미지의 품질사이의 적절한 상호 만족선에서 선택할 수 있다. 위키피디어에 소개된 샘플 사진으로 파일크기와 압축률 사이의 관계를 비교해 보도록 하자.

벡터 그래픽

위에 언급된 압축 방법들은 모두 픽셀단위의 bit pattern인 비트맵을 기본으로 하여 만들어진 방법이지만, 벡터 방식 표현은 전혀 다른 방식의 표현 방법이다. 벡터 그래픽은 픽셀단위로 정보를 저장하는 것이 아니라 선과 도형을 이용하여 이미지를 표현하는 형식이다. 따라서 선과 도형의 방향, 두께와 색상 같은 정보들이 저장되어 있다. 따라서 사진과 같은경우에는 적당하지 않고 라인아트나 만화 같은 경우에 매우 효율적으로 사용될 수 있다. 파일크기는 이미지 안에 있는 모습들이 얼마나 복잡하냐에 따라 다르지만 대부분의 라인아트 같은 경우에는 파일 크기가 매우 작아진다. GIF같은 비트맵 형시에서는 이미지 크기를 확대 혹은 축소를 하게 되면 인코딩을 다시 해야 하지만, 벡터 그래픽에서는 확대축소가 수학적으로 동적으로 계산되어 이루어지게 된다. 따라서 품질이 떨어지게 되는 경우가 없다.

파일크기와 이미지 크기의 자유로운 변환 때문에 웹에서 주로 사용하고 있는데 대표적인 것이 플래시(Flash) 이미지이다. 또한 SVG(Scalavle Vextor Graphics)도 웹에서 많이 사용되는 형식이다. 다음은 간단한 이미지에 대한 SVG파일의 내용을 보여주고 있다. 예를 들어 두번째 줄의 내용을 보면 사각형(rect)을 만드는데 시작점을 (80, 60)에서 시작해서 넓이와 높이를 각각 250으로 하고 코너각의 둥글기 정도, 색상, 가장자리 선의 색깔과 두께가 기술되어 있다.

소리 데이터의 인코딩

오디오(audio)란 소리나 음악이 표현되는 것을 말한다. 소리는 높이, 크기 그리고 음색으로 이루어져 있는데, 모두 진동수와 파형으로 구분이 된다. 오디오를 저장하는 표준 방법은 없지만 기본적으로 소리를 디지털 형식으로 변환하는 절차는 유사한데, 먼저 원래 아날로그인 오디오 파형을 디지털 데이터로 변환하고 그리고 bit pattern을 사용하여 저장한다. 아래 그림은 그 구체적인 과정을 보여주고 있다.

먼저 아날로그 신호가 샘플링된다. 샘플링은 시간에 따라 일정하게 간격으로 나누고 각 간격 마다 신호의 값을 측정하는 것이다. 그렇게 만들어진 샘플들은 양자화(quantization)된다. 양자화는 각 샘플에 정해진 값을 지정하는 것이다. 예를 들어 샘플 값이 23.4이고, 우리가 만약 6비트를 사용하여서 0부터 63까지 64가지 정수를 사용한다고 한다면 그 샘플 값을 23에 배정한다는 것이다. 그리고 그 양자화 된 값은 bit pattern으로 변경되는데, 예를 들면 23은 이진수로 바뀌어서 10111이 되어 저장된다.

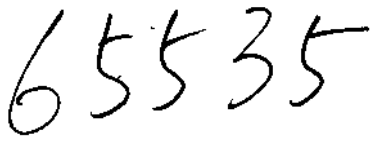
일반적으로 샘플링은 초당 40,000정도로 하면 사람의 귀에 거슬리지 않는 정도라고 한다. 즉 1초를 40,000개의 간격으로 쪼개어 그 각각에 디지털 값을 배정하게 되는 것이다. 만약 샘플링율을 더 늘린다면 저 좋은 품질의 소리를 얻을 수 있겠지만 어느 정도 이상이 되면 사람은 그 차이를 구별할 수 없게 된다. 양자화는 소리의 높낮이를 결정하게 되는데 만약 각 샘플당 8bits를 사용한다고 하면 256가지 소리 높낮이 정보를 표현하게 되는 것이다. 사람의 목소리만 주로 사용되는 전화품질에서는 8bit가 사용되며, 음악 CD에서는 16bits을 사용하여 65,536개의 전압값으로 소리를 표현한다. 하지만 아날로그 값을 양자화 하는 과정에서 일부 정보가 손실되기도 한다. 예를 들어 23.4의 값인데 23에 배정을 하여 0.4만큼의 음원 정보가 손실되는 것이다. 혹은 아주 낮은 음이나 아주 높은 음 같은 경우에는 표현하지 못하게 될 수도 있다. 특수한 상황에서는 24bits 양자화를 사용하기도 한다.

만약, 샘플링율이 초당 40,000이라고 하고 16bits 양자화 한다고 가정하면 10분짜리 노래는 몇 바이트를 차지하게 될까. 10분은 10*60=600초이므로 전체 크기는 600초x40,000x2바이트 = 48MB가 된다.
따라서 소리정보에 대해서도 압축 인코딩 방법이 필요하게 된다. 일반적으로 WAV 파일 형식이 압축되지 않은 거이고, 가장 일반적인 압축된 파일 형식은 MP3라고 할 수 있다.

MP3는 MPEG-2, audio layer 3 file의 줄임말이고, MPEG은 Moving Picture Express Group을 의미하는데 디지털 오디오와 비디오 압축에 대한 개발 표준을 협의하는 국제 위원회이다. MP3는 손실압축과 비손실압축 모두를 사용한다. 먼저 소리의 frequency 전체분포를 분석해서 인간 음향심리학(귀와 뇌 청각세포간의 관계에 대한 연구)에서 만들어진 수학모델과 비교하여 인간의 귀에 잘 들리지 않는 정보를 제거해 버린다. 그런 후에 호프만 인코딩과 같은 방법을 사용하더 추가적인 압축을 한다.

수치정보의 인코딩

수치정보인가, 숫자문자인가, 숫자이미지인가
아래의숫자를 보자. 이것을 어떻게 인코딩 할 것인가.



이것을 우리가 어떻게 인식하는가, 그리고 어떻게 사용할 것인가에 따라서 다음의 세 가지 방법을 생각해 볼 수가 있다.

1) 문자로 생각한다면, 위의 65535는 5개의 문자로 이루어져 있고, 각 문자는 아스키코드에 따라서 인코딩하면 모두 40bit가 필요하게 된다. Unicode로 한다면 80bit가 필요할 것이다. 다만, 이 경우에는 텍스트로만 사용할 수 있지 숫자의 계산에는 사용하지 못한다.
2) 이미지로 생각하여 표현할 수 있다. 즉 각 픽셀당 bit pattern으로 변환하여 표현할 수 있다. 이 경우에는 이미지상에 이 숫자가 표현되게 할 수 있다. 숫자나 문자로서의 정보는 가지지 못하게 된다.
3) 숫자로 생각한다면, 이진변환을 한다면 16bit로 표현할 수 있다. 이 경우에는 숫자의 연산을 할 수가 있다.

즉, 어떤 목적으로 사용하냐에 따라서 다르게 인코딩을 할 수 있다. 계산을 위해 필요한 것인지, 이미지로 보이게 하려고 하는 것인지, 텍스트로 보이게 하려고 하는것인지에 따라서 다른 인코딩이 사용될 수 있다. 우리가 '수치정보'라고 이야기 할때에는 '연산'을 염두해 두고 있는 것이며, 여기서는 수치정보의 인코딩에 대하여 간단하게 살펴보도록 하자.

이진수의 변환

우리가 앞서 배운 것 처럼 이진수의 형태는 각 칸마다 두 가지 정보 (1과 0)만 표현 할 수 있으며 그 칸들을 여러 개 위치시켜서 숫자를 표현하게 된다. 예를 들어서, 8비트를 사용한다면 각 비트마다 1또는 0으 두가지 경우를 표현하므로, 그 조합을 따지면 2^8=256 가지의 가른 경우를 표현할 수 있다. 그리고 그 256가지를 어떻게 8비트로 pattern을 만드느냐는 다음과 같이 간단히 처리할 수 있다.

즉, 만약 10진법 201을 이진법으로 만든다고 한다면, 위의 각자 고유 숫자가 써있는 카드를 한 번씩만 조합해서 어떻게 201을 만들 것인가를 생각 해 보면 된다. 큰것부터 차례로 체크해가면서 더해가면, 128+64+0+0+8+0+0+1=201 이 되고, 사용된카드는 1을 사용되지 않은 카드는 0이라고 생각하면, 11001001이 된다. 반대의 경우도 마찬가지이다. 만약 10001101이라면 1이 쓰여진 카드의 (즉 사용된 카드의) 고유 숫자만 더해주면 된다. 즉 128+0+0+0+8+4+0+1=141이 된다.
이진법과 십진법의 변환 방법은 중고등학교 때 이미 배웠을 것이며 각자마다 자신 있는 변환 방법을 알고 있을 것이다.

정수(Integer)의 표현

정수의 바이너리 인코딩은 얼핏 생각한다면 그냥 이진변환하기만 하면 되는 것에 불과할 수도 있지만, 실제로는 다음과 같이 생각해야 할 문제들이 있다.
1) 표현에 사용할 수 있는 bit수에 제한이 있기 때문에 (예를 들어 8bits) 표현할 수 있는 정수 숫자의 범위에 제한이 있다. 즉, 범위와 표현에 제한.
2) 정수의 음수는 어떻게 표현할 것인지의 문제가 있다.
3) 그렇게 표현 했을 때, 덧셈과 뺄셈과 같은 연산이 효율적으로 수행 될 수 있는지.
이제 소개될 여러 가지 정수 인코딩 방법은 위와 같은 고려사항에 따라서 생각해보도록 하자.

먼저 Unsigned 표현은 부호를 생각하지 말고 바로 이진수로 변환하는 것을 말한다. 만약 8bit를 사용한다면 0~255까지 표현이 가능하며, 16bits을 사용한다면 0~65,535의 표현이 가능하다. 하지만 그 번위를 벗어나는 숫자는 표현하지 못한다. 또한 음수

를 표현하지 못하므로 주로 횟수를 세는 카운터(counter)나 메모리의 주소할당 등에 사용이 된다.
이번에는 양수와 음수를 모두 표현하는 signed 표현에 대하여 생각해 보자. Signed-and-Magnitude 형식은 제일 첫 비트는 부호로(0은 양수, 1은 음수로) 사용하고 나머지 bits에 크기(magnitude)를 표현한다. 따라서 8bits인 경우에 첫 bit는 부호에 사용하고 나머지 7bits만 크기에 사용하므로 크기는 0~125까지만 표현이 된다. 각 부호에 대하여 그렇게 되므로 -125 ~ +125까지 표현이 가능하다. 만약에 8bits로 -9를 표현한다고 하면, 부호가 -이므로 제일 첫 비트는 1이 되고, 나머지 7bits는 9를 표현하면 된다. 즉 0001001가 되므로 부호비트까지 합치면 10001001이 된다. +9는 제일 첫 bit값만 0으로 바뀌면 된다. 즉, 00001001가 된다. 다음은 몇가지 다른 예를 보여주고 있다. 반대의 경우도 간단하다. 제일 왼편 비트의 값에 따라서 부호를 적고, 나머지 bits에 있는 이진값을 십진값으로 변환해 주면 된다.

십진수	8bit 사용	16bit 사용
+7	00000111	0000000000000111
-124	11111100	1000000001111100
+256	오버플로우 에러	0000000100000000

이방법은 십진수와 이진수를 상호 변환 하는 것이 매우 간단하다는 장점이 있는 반면에 문제들도 있는데 그중 하나는 십진수 0에 대한 표현이 두 가지가 나온다는 것이다. 즉, 8bits 사용의 경우에
+0 -> 00000000
-0 -> 10000000
의 두 가지 표현이 나와서 혼란을 초래한다. 또 덧셈이나 나눗셈의 연산이 부호 때문에 간단하지 않다. 따라서 연산보다는 이진과 십진의 상호 변한만 간단히 하는 경우에 주로 사용된다.

One's Complement(일의 보수) 표현방법은 조금 다른 규칙을 사용한다. 먼저 십진수가 양수인 경우에 그대로 이진수로 변환한다. 만약 그것이 음수인 경우에는 먼저 같은 크기의 양수에 대한 이진수를 먼저 구한 후, 그것의 1의 보수, 즉 0은 1로, 1은 0으로 바꿔치기를 한다. 예를 들어, +9인 경우에는 00001001로 표현하지만 -9인 경우에는 +9의 1의 보수, 즉 11110110이 된다. 즉, 같은 크기의 값이 서로 1의 보수 관계로 마주 보고 있는 모습이며, 제일 첫 비트가 0이면 양수, 1이면 음수임을 알 수 있다. 예를 들어, 01111100는 양수이므로 그대로 십진변환을 하면 +124가 되고, 10000011는 제일 왼쪽 비트가 1이므로 음수임을 알 수 있고 따라서 전체를 1의 보수를 취하면 01111100가 되어 이것을 십진변환시키면 124, 따라서 -124가 되는 것이다.

이 방법은 여전히 덧셈 뺄셈같은 연산이 간단하지 않고, 여전히 0이 두 가지가 존재하기 때문에 일반적으로 잘 사용은 되지 않으나, 통신에서 에러 검출에 사용 될 수도 있고 무엇보다도 다음에 나오는 Two's Complement(이의 보수)를 설명하기 위하여 미리 살펴 본것이다.

Two's Complement(이의 보수) 표현방법은 1의 보수에서처럼 0이 두가지로 표현되는 혼동과 그 혼동으로 인하여 연산이 복잡해 지게 되는 것을 막기 위하여 나온 방법이다. 양수인 경우에는 기존과 똑같이 이진 변환만 해주면 된다. 문제는 음수에 대한 표현인데 음수에 대하여 1을 더해주어 한칸씩 앞으로 당기는 모습을 보이는데 이렇게 하여 -0이 없어지고 그 자리에 -1이 들어가고, 1의 보수에서 -1의 자리에는 -2가 들어가는 식이다. 그러면 제일 마지막 음수는 빈칸이 되는데 여기에 -128이 새롭게 하나 더 들어간다. 이것을 그림으로 다시 보면서 이해해 보도록 하자. 만약에 8bit가 있다고 한다면, 8bit로 표현할 수 있는 range는 1's complement인 경우에 -127~+127 이 된다.

하지만 +0과 -0, 이렇게 두 개의 0이 있게 되고 bit pattern의 연산이 불편하다는 단점이 있다. 2's complement (2의 보수)는 0을 하나만 쓰게 해준다. 즉, 위의 그림에서 -0 대신에 -1이 올라오고, 모두 한 칸씩 올라가서 제일 아래에는 -128이 하나 더 추가될 수 있다. 따라서 2's complement에서는 range가 -128~+127이 된다.

이것은 다음과 같이 여러 가지로 표현이 가능한데 모든 같은 결과를 낸다. 예를 들어 어떤 음수의 정수 값을 생각해보자 (ex. -124).

- 먼저 양의 값만 이진 변환 시킨 후에 1의 보수를 취하고 1의 보수에서 1을 더해주면 2의 보수가 된다.
- 먼저 양의 값만 이진 변환 시킨 후에 1의 보수를 취하고 제일 오른쪽 비트에서 시작하여 왼쪽으로 한 칸씩 가면서 제일 처음 나오는 0까지만보수를 취해주고 나머지는 그대로 두는 것이다.
- 양의 값을 이진변환 한 후에 제일 오른쪽 비트에서 시작하여 왼쪽으로 가면서 첫번째 1까지는 그대로 두고 그 다음 비트부터 제일 왼쪽 비트까지 보수를 취한다.

이것을 -124의 예를 가시고 다시 살펴보자.

- 부호는 - 이고, 양의 값은 124이므로 124를 이진변환하면 01111100가 된다.
- 이것을 1의 보수 취하면 10000011가 된다.
  - 여기에 1을 더하면 10000100 가 되며,
  - 마찬가지로 제일 오른쪽 비트에서 왼쪽으로 가면서 처음 나오는 0까지만 보수를 취해주면 1000010가 된다.
- 이번에는 아예 124의 원래 이진변환값 01111100 에 대하여 제일 오른쪽 비트에서 시작하여 첫번째 1까지 즉, 100까지는 그냥 두고 그 다음부터 왼쪽 끝까지 보수를 취하면 10000100가 된다.

첫번 째 방법, 즉 1의 보수 취하여 1을 더한다는 것은 매우 개념적으로 설명한 것이고, 마지막 표현은 매우 알고리즘적으로, 즉 기계적으로 처리 될 수 있도록 표현 된 것이다. 다시 알고리즘적으로정리 하면,

- 만약 양수이면, 그대로 2진 변환
- 만약 음수이면, 그것의 양의 값을 2진 변환한 후에,
  - 제일 오른쪽 비트부터 시작하여 차례대로 제일 왼쪽 비트까지 가면서
    - 첫번째 1을 만나면 그 다음 비트부터 각 비트마다 보수를 취한다.

자 이제, 2의 보수에서는 0이 하나로만 표현되어 혼동이 없어지고, -128이라는 숫자 하나를 더 많이 표현할 수 있게 되었다. 아울러 뎃셈 뺄셈의 연산도 간단하게 처리 될 수가 있다는 것이 가장 큰 장점 중의 하나이다. 예를 들어

- (+3) + (+2) = 0011 + 0010 = 0101 = 5
- (−3) + (−2) = 1101 + 1110 = 1011 = −5 (즉, −0101) (왼쪽에 초과된 1은 무시)
- (+7) + (−5) = 0111 + 1011 = 0010 = 2

특히 2의 보수의 장점은 뺄셈도 덧셈과 똑같이 적용하면 된다는 것이다. 부호와 무관하게 같은 방법을 사용할 수 있으므로 덧셈만 생각하면 된다는 것이다. 결과적으로 2의 보수(complement)를 취하는 회로와 덧셈 회로만 있으면, 덧셈, 뺄셈, 곱셈, 나눗셈 모두를 수행할 수 있다. (곱셈은 덧셈을 반복하여 수행, 나눗셈은 뺄셈을 반복하여 수행하여 나머지가 음수가 되지 않을 때까지의 회수) 2의 보수 표현은 정수를 컴퓨터에서 표현하는 표준방법으로 사용되고 있다. 지금까지 살펴보았던 정수의 이진 인코딩 방법에 대하여 4bit 사용인 경우 몇가지 예를 보도록 하자.

Contents of Memory	Unsigned	Sign-and Magnitude	One's Complement	Two's Complement
0000	0	+0	+0	+0
0001	1	+1	+1	+1
0010	2	+2	+2	+2
0011	3	+3	+3	+3
0100	4	+4	+4	+4
0101	5	+5	+5	+5
0110	6	+6	+6	+6
0111	7	+7	+7	+7
1000	8	−0	−7	+8
1001	9	−1	−6	−7
1010	10	−2	−5	−6
1011	11	−3	−4	−5
1100	12	−4	−3	−4
1101	13	−5	−2	−3
1110	14	−6	−1	−2
1111	15	−7	−0	−1

또 하나의 정수 표현 방법이 있는데 Excess System 이라고 부른다. 연산은 사용하지 않고 간단하게 십진과 이진을 변환하는 경우에 주로 사용된다. 다음에 배우게 될 부동소숫점 표현에서 지수값을 인코딩할 때 사용된다. 만약 8bits를 사용하면 0~255까지 표현이 되는데 그중에 가운데 숫자, 즉 128 혹은 127을 매직숫자라고 정해서 그것을 기준으로 음수와 양수를 나누는 것이다. 만약 매직숫자를 128로 정하면 Excess_128 형식이라고 부른다. 아래의 그림에서 아래 칸이 실제 표현하려는 숫자이고 윗 칸이 이진 표현이 된다.

예를 들어, -25를 Excess_128로 하면 -25+128을 하여 +103이 되고 이것을 8bit 크기로 이진변환하면 01100111 가 된다. 반대로 만약에 Excess_128로 표현된 이진코드가 11111110이면 먼저 이것을 십진변환하여 254를 구한다. 그러면 254에서 매직넘버 128을 빼면 126가 된다.

부동소숫점(floating-point)의 표현

우리는 정수의 인코딩에 대하여 살펴 보았지만 이번에는 실수의 표현에 대하여 간단히 살펴보자. 실수는 12.345와같이 정수부분과 소수부분, 그리고 그들 사이에 소수점(.)이 있는 수를 이야기 한다. 실수를 표현 할 때 부동소수점(floating point) 표현방식은 소수점의 위치를 고정하지 않고 그 위치를 따로 표시하는 것으로, 가수(假數; mantissa)와 소수점의 위치를 표시하는 지수(指數; exponent)로 나누어 표현한다. 예를 들어 십진 실수 -0.064를 부동소숫점으로 표현하면 -64x10-3이 된다. 여기서 -64를 가수(mantissa), -3을 지수(exponent)라고 부른다.

먼저 일반적인 실수의 모습을 생각하면서 그 표현 방법을 생각하는 것보다 해보도록 하자. 물론 정수부분을 이진변화 하고, 소수부분을 이진변환 시키면 되긴 한다. 정수부분의 이진수 변환 방법은 앞서 각자의 고유 카드를 가지고 예를 들어 설명하였는데, 소수점 아래의 값들도 마찬가지로 설명될 수 있다.

먼저 이진수를 십진수로 변환하는 것부터 (쉬운 것부터!) 보도록 하자. 1001.1101이라고 한다면 정수부분은 1001이므로 위의 카드그림을 보고 적어 보면 8+0+0+1이되어 9가 되고, 소수부분은 1101이므로 1/2 + 1/4 + 0 + 1/16 이 되어 13/16=0.8125이 된다. 따라서 원래 이진수는 9.8125로 변환된다.

십진수를 이진수로 변환하는 것은 조금 더 복잡한 문제가 있다. 위의 그림을 자세히 살펴본다면 정수 부분은 0, 1, 2, 4, 8, 16, …, 2n-1 의 숫자(카드)들을 각 한번 씩만 사용 (또는 안사용)하여 조합하면 범위내의 (위그림에서는 0~2n-1) “어떠한” 정수라도 만들어 낼 수 있다. 하지만, 소수점 이하의 부분은 어떠한가. 1/2, 1/4, 1/8, 1/16, … 의 카드 각각을 (사용, 안사용) 조합하여 “어떠한” 숫자라도 만들어 낼 수 있는가? NO! 아래의 에를 보면 소수점 아래 이진수값 4자리에 대하여 모든 조합을 나열하여 보았다. 각각에 해당하는 십진수 값을 보면 연속된 실수값을 얻을 수 없다는 것을 알 수 있다. 예를 들어 0.2를 표현할 수 있는가. 없다. 만약 비트수를 늘인다면 예를 들어 100자리까지 만들면 되지 않을까 생각하지만, 그래도 그 사이에 빈 값이 존재 하기 때문이다. 그리고 무제한으로 비트 자리수를 늘릴 수도 없는 노릇이다. 그리고 아무리 자릿수를 늘리고 2의 마이너스 몇 승을 한다고 해서 표현하지 못하는 수가 존재 한다. 그것이 바로 십진수의 실수 0.1 (즉, 1/10) 이다. 실제로 1/10을 이진수로 변환하면 순환소수인 0.0001100110011001100…… (1100 이 계속 무한대로 반복). 따라서 2진수로는 가장 근사한 값이 사용되게 되는데 이것이 컴퓨터가 계산을 틀리게 할 수 있음. 예를 들어, 십진수인 0.1을 100번 반복해서 더하면 10이 아니라 9.9999999999999 가 나온다.

부동소수를 표현하는데 IEEE 표준에서는 single-precision과 double precision을 사용한다. Single은 32bits, double은 64bits를 사용한다. 32비트를 사용하는 Single precision의 예를 가지고 살펴보도록 하자. 32bits는 sign에 1bit, exponent(지수)에 8bits, 그리고 mantissa(가수)에 23bits를 사용한다.

- mantissa에는 1.xxxxxx 형태의 normalized representation을 사용한다.

예를 들어, 원래의 이진수가 1011.0011 이라면 제일 왼쪽의 1이 소수점 바로 왼쪽으로 올 때까지 오른쪽으로 shift 시킨다. 즉, 1.0110011 이 된다. (3번 sifted) 가수부가 23bit이지만 첫 번째 자리의 1이 생략되어 있기 때문에 실제로는 24bit의 값을 나타낸다.

- exponent는 Excess System 형식인 Excess-127을 사용한다.

그 이유는 sign bit를 사용하지 않고 음수를 나타내기 위하여서이다. Excess representation은 가장 중간 값을 0으로 간주한다. 그

래서 어떤 값이 있으면 그것을 가장 가운데 값으로 빼면, 그것의 excess representation 값이 된다. 즉, 8bits를 사용하면 0~255까지를 표현할 수 있는데, 그것의 가장 가운데 값, 즉 127을 0으로 간주하면 만약에 Excess_127 값이 255라면 255-127=128이 된다. 만약 100이라면 100-127=-27이 된다. 이렇게 해서 exponent(지수)에서는 2-127 ~ 2+128 까지를 표현할 수 있다. Exponent(지수)를 8bit를 사용하는 것을 Excess_127 이라고 부르고, 11bit를 사용하는 것을 Excess_1023이라고 한다. 이 지수값은 이진수 값에서는 2의 지수가 된다. 즉 만약 -5이라면 2-5가 된다.

Examples)
십진수 0.75를 single-precision floating-point representation으로 변환하여 보자.
답부터 이야기하면 0 01111110 10000000000000000000000 이다.

- Sign bit: 양수이므로 0.
- Exponent: 011111110 이것을 십진수로 바꾸면 126이 된다. 127-excess 방법을 사용하므로 126-127=-1, 즉 2-1
- Mantissa: 소수점 이상의 첫 번째 자리를 1로 하는 normalized 표현이므로 실제로는 2진수 1.10000..0 을 나타낸다. 이것은 1*20 + 1*2-1 이므로 1.5가 된다.

정리하면 +2-1 x 1.5 = 0.75 가 된다.

0.75를 1/2, 1/4, 1/8, 1/16.. 의 카드의 조합으로 표현하는 방법을 생각해보자. 먼저, 1/2=0.5를 사용하고, 두 번째 1/4=0.25를 합하면 된다. 즉 0.5+0.25=0.75. 즉. 0.11이 된다. 하지만, normalized representation을 사용하므로 소수점 왼쪽으로 1을 보내줘야 하기 때문에 1.1로 표현되고 그로 인해서 exponent에서 -1을 표시하여 주면 된다. 즉 exponent는 -1이 되고 mantissa에서는 10000.. 로 표현하면 된다.

몇가지 사례를 가지고 배운 것을 확인해 보도록 하자.

Number	sign	Exponent	Mantissa
−22 × 1.11000011	1	10000001	11000011000000000000000
+2−6 × 1.11001	0	01111001	11001000000000000000000

.끝.