**code\_1 – 최대공약수 구하기**

최대공약수를 구하는 알고리즘을 c++로 구현했을 때 다음과 같다.

|  |
| --- |
| #include<vector>  #include<iostream>  using namespace std;  vector<int> gcdlcm(int a,int b)  {  vector<int> answer;  int copy\_a = a, copy\_b = b;    while (b) {  if (a > b) a -= b;  else b -= a;  }  answer.push\_back(a);  answer.push\_back(copy\_a\*copy\_b / a);  return answer;  }  int main()  {  int a=3, b=12;  vector<int> testAnswer = gcdlcm(a,b);  cout<<testAnswer[0]<<" "<<testAnswer[1];  } |

최대 공약수를 구할 때 알고리즘적인 접근방식은 유클리드 호제법이 있다.

큰 것에서 작은 것을 나누고, 나머지와 작은 수를 나눠주는 계산을 반복하여 작은 수가 0이 됐을 때 반대쪽 변수에 저장된 값이 최대공약수가 되는 알고리즘이다.

이를 바탕으로 어셈블리어를 작성해보자.

|  |
| --- |
| section .data  array dw 70, 8  len\_of\_array equ $ - array  section .bss  result resb 2  num\_string resb 6  ; print 매크로 생성  %macro print 2  mov rax, 1  mov rdi, 1  mov rsi, %1  mov rdx, %2  syscall  %endmacro  section .text  global \_start  \_start: ; 최대공약수 계산  movzx rax, word [array] ; 첫 번째 변수를 rax에 로드 (word to dword 확장)  movzx rbx, word [array + 2] ; 두 번째 변수를 rbx에 로드 (word to dword 확장)  continue\_loop:  cmp rax, rbx  jb xchg\_loop  ja div\_loop  je done  xchg\_loop:  xchg rax, rbx  test rbx, rbx  jz done  jmp continue\_loop  div\_loop:  xor rdx, rdx  div rbx  mov rax, rdx  jmp continue\_loop  done:  mov [result], rax ; 최대공약수 출력  mov rsi, result  movzx eax, word [rsi]  call print\_number  ; 프로그램 종료  mov rax, 60  xor rdi, rdi  syscall  print\_number:  mov rdi, num\_string + 5  mov byte [rdi], 0  mov rcx, 10  convert\_loop:  dec rdi  xor rdx, rdx  div rcx  add dl, '0'  mov [rdi], dl  test rax, rax  jnz convert\_loop  mov rsi, rdi  lea rdx, [num\_string + 6]  sub rdx, rsi  print rsi, rdx  ret |

작성된 코드는 다음과 같은데, 크게 네 부분으로 나눠서 설명할 수 있다.

1. print macro 생성과 프로그램 종료
2. section .data, section .bss
3. continue\_loop
4. xchg\_loop, div\_loop이다.

<1> print macro

%macro로 시작하여 %endmacro 를 사용하면 분기점 생성과는 다르게 새로운 명령어를 생성할 수 있어서 반복적인 사용이 예상될 경우 선언할 수 있다.

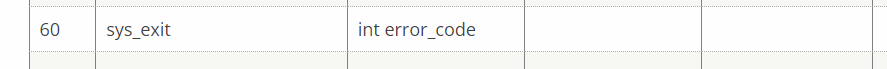
%macro print 2 에서 print는 함수 호출 이름이고 2는 입력받는 인자의 수를 의미한다.

그리고 rsi, rdx에는 %1, %2이 작성돼있는데, 함수를 호출하며 입력받는 첫 번쨰, 두 번째 값을 숫자에 맞춰 입력받게 된다.

이때 system cal을 이해하기 위해서는 x86의 syscall table을 참고해야한다.

64bit system call table을 참고했을 때텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명



|  |
| --- |
| %macro print 2  mov rax, 1  mov rdi, 1  mov rsi, %1  mov rdx, %2  syscall  %endmacro  ; 프로그램 종료  mov rax, 60  xor rdi, rdi  syscall |

rax에 1을 입력받았으므로 sys\_read를 호출한 것이다. 이때 rdi는 file description 값인데

standard input값은 0, standard output 값은 1, standard error 값은 2이다.

rsi는 출력할 text를 입력받고, rdx는 출력할 text의 개수를 반영한다.

또한 rax에 60을 입력받을 때는 sys\_exit이 호출되어 시스템을 종료한다.

<2> section .data, section .bss

|  |
| --- |
| section .data  array dw 70, 8  len\_of\_array equ $ - array |

실험에 사용될 데이터를 array배열에서 받는다. len\_of\_array는 equ함수를 써서

[ 현재 포인터 위치 – array주소 시작위치 ] 를 하여 array의 크기를 반영한다.

|  |
| --- |
| result resb 2  num\_string resb 6 |

코드에서는 result, num\_string 변수에 각각 2byte, 6byte를 할당받았다. 해당 값은 메모리의 시작 주소값을 갖는다.

<3> continue\_loop

|  |
| --- |
| continue\_loop:  cmp rax, rbx  jb xchg\_loop  ja div\_loop  je done |

continue\_loop라는 분기점 하나를 생성한다. cmp 명령어를 통해 rax, rdx에 저장된 값을 비교한 후, 오른쪽 값이 크다면 jb를 통해 xchg\_loop로 보내지고, 왼쪽 값이 크다면 div\_loop로 보내지고, 같은 값을 가진다면 자기 자신이 최대공약수이므로 done으로 보내진다.

<4> xchg\_loop, div\_loop

|  |
| --- |
| xchg\_loop:  xchg rax, rbx  test rbx, rbx  jz done  jmp continue\_loop  div\_loop:  xor rdx, rdx  div rbx  mov rax, rdx  jmp continue\_loop  done:  mov [result], rax ; 최대공약수 출력  mov rsi, result  movzx eax, word [rsi]  call print\_number  ; 프로그램 종료  mov rax, 60  xor rdi, rdi  syscall |

rbx값이 더 컸을 경우 jb를 통해 xchg\_loop로 보내져 좌우 값이 xchg를 통해 위치를 바꾸게 된다. test함수는 rbx와 rbx의 모든 bit를 and연산하는데, rbx의 모든 값이 0이라면 test의 결과값은0이 된다. 즉, 모든 값이 0인지 검사하기 위한 기교이다.

만일 test의 결과가 0이라면 jz 조건을 통해 done으로 보내지고, 0이 아니라면 아직 연산을 마무리하지 않았기 때문에 jmp를 통해 continue\_loop로 돌아간다. 이 함수의 종료 조건은 b=0이 되는 것이다.

div\_loop는 rax>rbx조건일 때 rax/rbx를 하여 몫은 rax, 나머지는 rdx에 저장된다. 이 연산에서 사용되는 것은 나머지이기 때문에 rdx의 값을 rax로 옮겨준 뒤 continue)\_loop로 보내진다.

최종적으로 done 분기점으로 보내졌을 때 rax에 저장된 값을 result 메모리에 저장하고 print\_number분기점으로 결과값을 보내며 함수가 종료된다.

마지막으로 출력부분에 해당하는 print\_number를 살펴보자.

|  |
| --- |
| print\_number:  mov rdi, num\_string + 5  mov byte [rdi], 0  mov rcx, 10  convert\_loop:  dec rdi  xor rdx, rdx  div rcx  add dl, '0'  mov [rdi], dl  test rax, rax  jnz convert\_loop  mov rsi, rdi  lea rdx, [num\_string + 6]  sub rdx, rsi  print rsi, rdx  ret |

print\_number는 최종적으로 rax에 저장된 값을 출력하는 분기점이다.

num\_string에서 5byte공간을 할당하기 위해 5byte 앞으로 이동한 주소를 rdi에 저장해준 뒤

0값을 rdi 주소에 저장해준다. 이 방법은 문자열이 끝나는 부분에 0을 확실히 표시하는 것이다.

한편, 이때 rcx에 저장되는 10은 10진수를 표현하기위해 사용될 것이며, xor rdx,rdx는 rdx를 0으로 초기화하는 기교이다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명conver\_loop에 들어가면 rdi는 한 칸 이동하여 문자열의 마지막 부분에 입력된 값을 가리킨다.

div rcx는 rax/rcx로 동작되며, rax는 다음 숫자를, rdx에는 문자열로 변환시킨 정보가 저장되며 rdi 위치에 해당 문자가 저장된다.

예를 들어 rax에 123이 입력됐다고 치자, div rcx를 통해 rax에는 몫인 12, rdx에는 나머지 3이 저장된다. 한편, rdi는 num\_string의 가장 마지막 부분을 가리키고 있다. add dl “0”를 통해 ascii코드 문자0으로부터 rdx만큼 떨어진 부분으로 다가가면 rdx에 저장된 숫자와 동일한 숫자에 해당하는 문자를 얻을 수 있게 된다. 이 문자를 rdi에 저장한뒤 rax가 0이 아니라면, 즉, 여전히 문자열 위에서 정보를 검색 중이라면 rdi를 한 칸 더 감소시켜 rax에 저장된 12를 rcx인 10과 나누는 과정을 반복한다. 이 과정을 숫자를 뒤에서부터 하나씩 문자열로 변형하는 과정을 거치는 것이다.그러면 rax에 저장된 모든 숫자는 num\_string에 문자열로 저장되고 시스템이 종료된다.

**code\_1 – test case**

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 라인, 화이트이(가) 표시된 사진

자동 생성된 설명

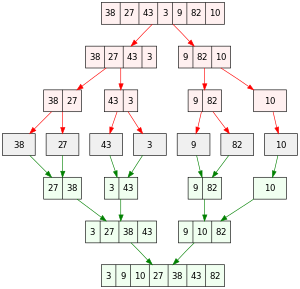
텍스트, 폰트, 라인, 화이트이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

**code\_2 – merge sort**



merge sort는 가장 작은 단위로 분할한 뒤, 각 분할에서 왼쪽 블럭과 오른쪽 블럭 각각의 가장 작은 값과 비교하여 재정렬하는 과정을 반복하는 과정을 가져야한다.

이를 수행하기 위해서는 분할되는 각 값의 왼쪽 오른쪽 블럭의 주소 위치를 기억해야하며, 각 주소에 저장된 값이 변경되는 경우를 추적해야한다.

이 과정을 c++로 구현했을 때

|  |
| --- |
| #include <iostream>  using namespace std;  int a[10] = { 5,879456,4,3,5412,2,1,457,45120,6523};  //합치기  void m\_sort(int list[], int l, int m, int r)  {  int i, j, k;  i = l;  j = m + 1;  k = l;  int temp[11];  //왼쪽에서 정렬된 배열과 오른쪽에서 정렬된 배열을 합침  //둘중 하나가 끝까지 올때 까지 합침  while (i <= m && j <= r)  {  if (list[i] <= list[j])  temp[k++] = list[i++];  else  temp[k++] = list[j++];  }  //오른쪽 배열에 임시 배열로 추가할 수가 남았으면 오른쪽 배열을 임시 배열로 복사  if (i > m)  {  for (int x = j; x <= r; x++)  temp[k++] = list[x];  }  //왼쪽 배열에 임시 배열로 추가할 수가 남았으면 왼쪽 배열을 임시 배열로 복사  else  {  for (int x = i; x <= m; x++)  temp[k++] = list[x];  }  //본 배열으로 복귀  for (int x = l; x <= r; x++)  list[x] = temp[x];  }  void merge(int list[], int l, int r)  {  int mid;  if (l < r)  {  mid = (l + r) / 2;  merge(list, l, mid);  merge(list, mid + 1, r);  m\_sort(list, l, mid, r);  }  }  int main()  {  merge(a,0,9);  for (int i = 0; i < 10; i++)  {  cout << a[i] << " ";  }  return 0;  } |

이렇게 재귀함수로 표현할 수 있다.

특히나

|  |
| --- |
| void merge(int list[], int l, int r)  {  int mid;  if (l < r)  {  mid = (l + r) / 2;  merge(list, l, mid);  merge(list, mid + 1, r);  m\_sort(list, l, mid, r);  }  } |

이렇게 한 층씩 merge를 해나가는 부분은 재귀함수에서 탈출했을 때 이미 블럭 내부는 배열이 정리되어있음을 가정하고 sorting하는 함수를 잘 작성해야한다.

한편, assembly 코드로 작성할 땐, c++과 달리 for문의 카운터 변수를 이용할 수 없기 때문에 pop push*를 적절히 이용하여 재귀함수 탈출 후 돌아왔을 때 배열 내의 값은 변경되고, 배열 주소 값은 회복되는 알고리즘을* 작성해야한다.

작성된 코드는 다음과 같다.

|  |
| --- |
| section .data  array db 9,8,7,6,3,4,2,1,5,10,13  len\_of\_array equ $-array  section .bss  result resb 2  num\_string resb 8  %macro print 2  mov rax, 1  mov rdi, 1  mov rsi, %1  mov rdx, %2  syscall  %endmacro  section .text  global \_start  \_start:  mov rsi, array  mov rax, len\_of\_array  mov r8, result  call divide ; rsi와 rax를 적절히 재귀함수로 호출하여 정렬하자  mov r12, len\_of\_array ; 최종적으로 저장된 array값 반복인자  loop:  mov rax, len\_of\_array  sub rax, r12  lea rdi, [array]  add rdi, rax  movzx rax, byte [rdi]  call print\_number  ; 프로그램 종료  mov rax, 60  xor rdi, rdi  syscall  print\_number:  mov rdi, num\_string + 5  mov byte [rdi], ' '  mov rcx, 10    convert\_loop:  dec rdi  xor rdx, rdx  div rcx  add dl, '0'  mov [rdi], dl  test rax, rax  jnz convert\_loop    mov rsi, rdi  lea rdx, [num\_string + 6]  sub rdx, rsi    print rsi, rdx    dec r12  test r12, r12  jnz loop  ret  divide:  cmp rax, 1  jbe done  mov rdx, rax  push rax  shr rax, 1  mov rcx, rax  sub rdx, rcx  lea rdi, [rsi+rcx]    push rcx  push rdx  push rdi  push rsi  mov rax, rcx ; 포인터 위치 update  call divide ; 왼쪽 블럭 접근    pop rsi  pop rdi  pop rdx  pop rcx  pop rax    push rax  push rcx  push rdx  push rdi  push rsi  mov rsi, rdi ; 포인터 위치 update  mov rax, rdx  call divide ; 오른쪽 블럭 접근    pop rsi  pop rdi  pop rdx  pop rcx  pop rax  push rax  push rcx  push rdx  push rdi  push rsi  push r8  call merge ; 정렬된 블럭 merge    pop r8  pop rsi  pop rdi  pop rdx  pop rcx  pop rax    mov rdi, rsi  mov rsi, r8  mov rcx, rax  rep movsb ; sort된 블럭 갱신  jmp done  merge:  movzx r9, byte[rsi]  movzx r10, byte[rdi]  cmp r9, r10  jbe select\_left  ja select\_right  ret    select\_left:  mov [r8], r9  inc r8  inc rsi  dec rcx  cmp rcx, 0  ja merge  jmp release\_right  select\_right:  mov [r8], r10  inc r8  inc rdi  dec rdx  cmp rdx, 0  ja merge  jmp release\_left  release\_left:  mov rdi, r8  rep movsb  jmp done  release\_right:  mov rsi, rdi  mov rdi, r8  mov rcx, rdx  rep movsb  jmp done  done:  ret |

이 코드에서 중요한 분기점은 divide와 merge이다.

<1> divide

|  |
| --- |
| divide:  cmp rax, 1  jbe done  mov rdx, rax  push rax  shr rax, 1  mov rcx, rax  sub rdx, rcx  lea rdi, [rsi+rcx]    push rcx  push rdx  push rdi  push rsi  mov rax, rcx ; 포인터 위치 update  call divide ; 왼쪽 블럭 접근    pop rsi  pop rdi  pop rdx  pop rcx  pop rax    push rax  push rcx  push rdx  push rdi  push rsi  mov rsi, rdi ; 포인터 위치 update  mov rax, rdx  call divide ; 오른쪽 블럭 접근    pop rsi  pop rdi  pop rdx  pop rcx  pop rax  push rax  push rcx  push rdx  push rdi  push rsi  push r8  call merge ; 정렬된 블럭 merge    pop r8  pop rsi  pop rdi  pop rdx  pop rcx  pop rax    mov rdi, rsi  mov rsi, r8  mov rcx, rax  rep movsb ; sort된 블럭 갱신  jmp done |

divide 분기점을 보면 call divide를 벗어났을 때 pop을 통해 모든 값을 회복한다.

divide loop를 돌 때 rax에 저장된 배열 길이가 1보다 크다면 레지스터 값들에 업데이트를 하는데, 이때, rsi, rdi, rdx, rcx, rax에 저장된 값은 각각 array의 시작주소(=왼쪽 블럭 시작 주소), 블럭의 중앙 주소(=오른족 블럭 시작 주소), 오른쪽 블럭의 요소 개수, 왼쪽 블럭의 요소 개수, 블럭 요소 개수(data 전달을 위한 레지스터)로 업데이트를 하며 array 길이를 나타내는 rax레지스터는 rcx인 블럭 요소 절반 개수로 업데이트되면서 왼쪽 정렬 loop를 마치게 된다.

한편 shr 명령어는 bit shift 연산자여서 //2 하여 2로 나눈 몫을 구하는 효과를 볼 수 있다 예를들어 5 (10)=101 (2)을 shr한다면 2 (10)=10 (2)이 된다.

그리고 오른쪽 정렬 loop를 위해서 rdi에 저장된 블럭의 중앙 주소를 블럭 시작 주소인 rsi에 전달하고, 기존의 블럭의 요소의 개수였던 rdx를 rax에 복원시켜주면서 해당 층의 블럭 요소 개수, 블럭 시작 주소, 블럭 중앙 주소를 복원시킬 수 있다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

<2> merge

|  |
| --- |
| merge:  movzx r9, byte[rsi]  movzx r10, byte[rdi]  cmp r9, r10  jbe select\_left  ja select\_right  ret    select\_left:  mov [r8], r9  inc r8  inc rsi  dec rcx  cmp rcx, 0  ja merge  jmp release\_right  select\_right:  mov [r8], r10  inc r8  inc rdi  dec rdx  cmp rdx, 0  ja merge  jmp release\_left  release\_left:  mov rdi, r8  rep movsb  jmp done  release\_right:  mov rsi, rdi  mov rdi, r8  mov rcx, rdx  rep movsb  jmp done  done:  ret |

divide를 마친 후 merge에 도달했을 때 rsi는 왼쪽 블럭의 시작 주소, rdi는 오른쪽 블럭의 시작 주소를 저장하고 있다.

movzx는 원래 크기보다 작은 데이터를 더 큰 레지스터에 로드하고 확장하는 작업을 한다. 왼쪽 블럭의 값을 r9, 오른쪽 블럭의 값을 r10에 저장하여 비교한 뒤 결과에 따라 select\_left, select\_right로 보내진다.

스크린샷, 텍스트, 직사각형, 디자인이(가) 표시된 사진

자동 생성된 설명일반성을 잃지 않고 두 분기점의 이용은 동일하므로 select\_left만 살펴보자. 최종 결과를 저장하는 r8 레지스터에 왼쪽 블럭 값인 r9을 저장하고 다음 입력값을 받기 위해 r8 주소를 한 칸 이동시킨 뒤 왼쪽 블럭 시작 주소인 rsi를 한 칸 이동시킨다. 이때 rcx는 왼쪽 블럭의 요소 개수를 가리키므로 하나 감소시킨다. 이때 더이상 왼쪽에 남은 요소가 없다면 release\_right를 실행시킨다.

이때 rep movsb라는 명령어를 사용하는데, rsi에 시작 주소, rdi에 목적 주소, rcx에 카운터 인자를 준 후 명령을 실행하면 카운터 수만큼 소스에서 목적지로 바이트를 복사한다.

즉, 왼쪽에서 이미 모든 값을 뽑아냈다면 오른쪽 블럭에서 나머지 모든 값을 뽑아낸다는 뜻이 된다.

그리고 done 분기점으로 보내져 ret을 통해 최초로 호출된 지점인 call merge로 돌아간다.

|  |
| --- |
| \_start:  mov rsi, array  mov rax, len\_of\_array  mov r8, result  call divide ; rsi와 rax를 적절히 재귀함수로 호출하여 정렬하자  mov r12, len\_of\_array ; 최종적으로 저장된 array값 반복인자 |

divide 함수를 적절하게 정의한 덕분에 \_start부분에서 짧은 코드로 가시성을 확보할 수 있게 되었다. 이제 r8에 순차적으로 저장된 배열 값을 출력하기 위해 print 매크로를 요소에 하나씩 접근하여 실행해주어야 한다.

|  |
| --- |
| loop:  mov rax, len\_of\_array  sub rax, r12  lea rdi, [array]  add rdi, rax  movzx rax, byte [rdi]  call print\_number |

여기서 call print\_number는 1번 문제에서 사용한 것과 동일하므로 설명을 생략한다.

한편 print rsi, rdx를 해줄 때마다 dec r12를 해주어 r8의 다음 요소에 접근할 수 있도록 rax값을 조정해준다.

**code\_2 – test case**









**code\_3 – 문자열 압축**

해당 문제를 이해하기 위해 c++로 작성하여 코드의 흐름을 살펴보자.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  int intlen(int n) {  int ret = 0;  while(n) {  ret++;  n /= 10;  }  return ret;  }  int main() {  const char \*s = "abcabcabab";  int answer = strlen(s);  int len\_s = strlen(s);  for (int len = 1; len <= len\_s / 2; len++) {  int now = len + len\_s % len;  int cnt = 0;  for (int i = 1; i < len\_s / len; i++) {  int flag = 1;  for (int j = i \* len; j < (i + 1) \* len; j++) {  if (s[j] != s[j - len]) {  flag = 0;  break;  }  }  if (flag) cnt++;  else {    if (cnt) {  now += intlen(cnt + 1);  cnt = 0;  }  now += len;  }  }  if (cnt) {  now += intlen(cnt + 1);  }  answer = answer < now ? answer : now; // min function  }  printf("Compressed length: %d\n", answer);  return 0;  } |

이 알고리즘에서 중점적으로 고민했던 부분은 세 가지다.

<.1> 문자열을 비교하는 chunk의 크기 최댓값은 문자열/2 를 넘지못하며, chunk는 1~길이/2 값을 가진다.

<2> chunk가 임의의 크기로 정해졌을 때, (1)다음 문자열이 동일하지 않은 경우, (2)다음 문자열이 동일하지만 이전에 동일한 적이 없었던 경우, (3)다음 문자열이 동일하지만 이전에 동일한 적이 있었떤 경우 세 가지로 나눠서 최종 문자열의 길이를 반영해줘야한다.

<3> 각 chunk의 압축 결과를 저장하는 공간이 필요하며, 압축 결과를 낼 때마다 이전의 결과와 비교하여 최솟값을 갱신한다.

그래서 <1> <3>문자열을 비교하는 chunk 크기를 모두 반영하고 최솟값을 찾아내는 부분과 <2> 임의의 chunk 에서 문자열을 비교하는 함수를 각각 정의한 후 합치는 과정을 통해 결과를 내었다.

|  |
| --- |
| section .data  array db "xababcdcdababcdcd"  len equ $ - array  space db ' '  section .bss  result resb 2  num\_string resb 2 ; 'a'와 null-terminator 저장  section .text  global \_start  %macro print\_int 1  mov rax, %1  mov rcx, 10  lea rdi, [num\_string + 2]  .convert\_digit:  ; rax를 rcx로 나눠서 각 자리 숫자를 아스키 문자로 변환  dec rdi ; rdi를 감소시켜 다음 문자 위치로 이동  xor rdx, rdx ; rdx 레지스터를 0으로 초기화  div rcx ; rax를 rcx(10)로 나누기 (몫은 rax, 나머지는 rdx에 저장)  add dl, '0' ; 나머지 값을 아스키 코드 문자로 변환  mov [rdi], dl ; 변환된 문자를 메모리에 저장  test rax, rax ; rax가 0인지 확인 (몫이 0인지 확인)  jnz .convert\_digit ; rax가 0이 아니면 반복  lea rsi, [rdi]  lea rdx, [num\_string + 2]  sub rdx, rdi  print rsi, rdx  %endmacro  %macro print 2  mov rax, 1 ; sys\_write  mov rdi, 1 ; stdout  mov rsi, %1 ; 주소  mov rdx, %2 ; 길이  syscall  %endmacro  \_start:  mov r14, 500 ; 최종 결과 저장  mov r9, len ; 배열 길이 로드  push r9  cmp r9, 1 ; 문자열 1개면 1 출력  je out  xor rdx, rdx ; 문자열 / 2 저장  mov rax, r9  mov rbx, 2  div rbx  mov r8, rax ;r8에 chunk 크기 저장    call chunk\_loop  print\_int r14    system\_out:  mov rax, 60 ; 시스템 종료  xor rdi, rdi  syscall    chunk\_loop:  xor r13, r13 ; equal 반복인자  xor r12, r12 ; 중간 결과값  sub r9, r8  lea r15, [array] ; 원본 위치  lea r10, [num\_string] ; 목표 위치  add r12, r8 ; 초기값 chunk 더해주기  call compare\_part\_loop ; 작업 시작  mov r9, len ; r9 복원    cmp r14, r12  jbe skip\_update  mov r14, r12  skip\_update:  dec r8  cmp r8, 1  jae chunk\_loop  jb done\_  compare\_part\_loop:  cmp r9, r8  jb back  call get\_word ; r10에 비교할 문자 저장, r10에 알파벳 저장  mov rsi, r15 ; 이동한 원본 위치 "(ab)abcdcde" -> "ab(ab)cdcde"  mov rdi, r10 ; chunk 추출한 문자 "ab"  xor rcx, rcx  add rsi, r8 ; 원본은 chunk칸 뒤의 글자와 비교  compare\_loop:  cmp rcx, r8 ;rcx(add), r8(int)  jae equal  xor rax, rax  xor rbx, rbx  movzx rax, byte[rsi] ; rax, rbx ,rcx, rdx, rsi rdi  movzx rbx, byte[rdi]  cmp rax, rbx  jne not\_equal  inc rsi  inc rdi  inc rcx  jmp compare\_loop  equal:  sub r9, r8 ; r9 = 1  cmp r13, 0  jbe equal\_first  ja equal\_repeat  equal\_first:  add r12, 1  add r15, r8 ; 다음 chunk로 이동 후 sampling  mov r13, 1 ; 1번 이상 반복 check  jmp compare\_part\_loop  equal\_repeat:  add r15, r8  jmp compare\_part\_loop    not\_equal:  add r15, r8  add r12, r8  mov r13, 0  sub r9, r8  jmp compare\_part\_loop  get\_word:  push rax  push rcx  xor rax, rax  word\_loop:  cmp rax, r8  jae done  movzx rcx, byte [r15 + rax] ; 첫 번째 바이트 로드  mov [r10 + rax], rcx ; num\_string에 저장  inc rax  jmp word\_loop  done:  pop rcx  pop rax  ret  done\_:  ret    back:  add r12, r9 ; 남은 문자 개수  ret  out:  mov rsi, 1  print\_int rsi  mov rax, 60 ; 시스템 종료  xor rdi, rdi  syscall |

이 코드에서 <1><3> 문자열을 비교하는 chunk 크기를 모두 반영하고 최솟값을 찾아내는 부분의 구현은 아래와 같다.

한편 문자열의 개수가 1이 경우, 추가적인 알고리즘이 필요없고 1을 출력하면 되므로 \_start 머리부분에 따로 분기점을 생성하였다.

|  |
| --- |
| chunk\_loop:  xor r13, r13 ; equal 반복인자  xor r12, r12 ; 중간 결과값  sub r9, r8  lea r15, [array] ; 원본 위치  lea r10, [num\_string] ; 목표 위치  add r12, r8 ; 초기값 chunk 더해주기  call compare\_part\_loop ; 작업 시작  mov r9, len ; r9 복원    cmp r14, r12  jbe skip\_update  mov r14, r12  skip\_update:  dec r8  cmp r8, 1  jae chunk\_loop  jb done\_ |

r14에는 최종 압축 업데이트 값,

r12에는 chunk별 압축 결과,

r8에는 chunk size,

r9에는 아직 작업하지 않은 문자열의 개수,

r13에는 이전에 동일한 문자열을 이미 만났는지 여부를 판단하는 인자로 정의했다.

r15에는 array의 시작 주소 ( 원본 array 보호를 위함 )

r10에는 다음 chunk의 array 주소를 저장하였다.

문자열 길이에서 /2를 한 뒤 chunk size 최댓값으로부터 r9을 dec를 하여 모든 가능한 chunk size를 탐색하는 chunk\_loop <1>를 정의한다.

compare\_part\_loop를 호출하여 임의의 chunk의 압축 결과를 내는 <2> 함수를 이용하면 r9은 값이 변형되므로 원본 문자열 길이로 복원하는 과정이 필요하다.

그 위 최종 결과 r14,, chunk별 결과 r12를 비교하여 업데이트 여부를 결정하는 skip\_update <3>를 정의한다.

한편, 두 번째 chunk 압축 값은 첫 번째 압축 값과 비교하면 되지만, 첫 번째 압축 값은 비교할 대상이 없으므로 알고리즘을 따로 정의해줘야하지만, 충분히 큰 수인 500을 0번 째 압축 결과로 제시한다면 추가적인 알고리즘 정의 없이 원활하게 동작할 수 있다.

또한 chunk size가 결정되면 이미 문자열 압축 길이는 chunk size를 포함하기 때문에 add r12, r8을 해줘야한다.

그럼 chunk별 문자열을 압축하는 알고리즘인 compare\_loop<2>를 살펴보도록 하자.

|  |
| --- |
| compare\_part\_loop:  cmp r9, r8  jb back  call get\_word ; r10에 비교할 문자 저장, r10에 알파벳 저장  mov rsi, r15 ; 이동한 원본 위치 "(ab)abcdcde" -> "ab(ab)cdcde"  mov rdi, r10 ; chunk 추출한 문자 "ab"  xor rcx, rcx  add rsi, r8 ; 원본은 chunk칸 뒤의 글자와 비교  compare\_loop:  cmp rcx, r8 ;rcx(add), r8(int)  jae equal  xor rax, rax  xor rbx, rbx  movzx rax, byte[rsi] ; rax, rbx ,rcx, rdx, rsi rdi  movzx rbx, byte[rdi]  cmp rax, rbx  jne not\_equal  inc rsi  inc rdi  inc rcx  jmp compare\_loop  equal:  sub r9, r8 ; r9 = 1  cmp r13, 0  jbe equal\_first  ja equal\_repeat  equal\_first:  add r12, 1  add r15, r8 ; 다음 chunk로 이동 후 sampling  mov r13, 1 ; 1번 이상 반복 check  jmp compare\_part\_loop  equal\_repeat:  add r15, r8  jmp compare\_part\_loop  not\_equal:  add r15, r8  add r12, r8  mov r13, 0  sub r9, r8  jmp compare\_part\_loop |

이때 get word 함수의 구현을 먼저 살펴봐야한다.

|  |
| --- |
| get\_word:  push rax  push rcx  xor rax, rax  word\_loop:  cmp rax, r8  jae done  movzx rcx, byte [r15 + rax] ; 첫 번째 바이트 로드  mov [r10 + rax], rcx ; num\_string에 저장  inc rax  jmp word\_loop  done:  pop rcx  pop rax  ret |

get\_word는 rax가 0일때부터 시작하여 chunk 사이즈만큼 문자열을 그대로 복사해온다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

그 다음

mov rsi, r15

mov rdi, r10

add rsi, r8

를 통해

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

원본 문자열의 복제본인 r15의 주소값을 나타내는 rsi의 주소를 chunk만큼 옮긴 다음 rdi와 한 바이트씩 비교하면 앞의 문자열과 뒤의 문자열이 동일한지 점검할 수 있다.

compare\_loop는 문자열이 동일한지 점검하는 분기점이다. 0으로 초기화된 rcx와 r8(=chunk)를 loop로 비교하는데, 동일하지 않은 문자열을 찾은 경우 not\_equal 분기점으로 가게 되고, chunk size를 넘어갈 때까지 문자열이 동일했다면 동일한 문자열로 볼 수 있기 때문에 equal으로 보내진다. 앞에서 한번 언급한 바와 같이, equal에도 이전에 equal이 있었는지, 없었는지를 한번 더 판단해야한다. cmp r13, 0 를 통해 이전에 equal한 문자열이 없었다면 equal\_first: 분기점으로 보내져서 문자열 압축 결과 r12에 1을 더해주게 되고, equal한 문자열이 이미 있었다면 r12에 아무런 변화가 없다.

한편 equal first에서는 이제 equal한 문자를 한번 만났기 때문에 r13을 1으로 업데이트 해줘야하고, not\_equal에서는 이제 equal한지 다시 검사를 해주어야하므로 r13을 0으로 초기화해준다.

추가적으로 가장 상단에 구현된

cmp r9, r8

jb back

를 짚고 넘어가자면, 만일 8개의 문자열에서 chunk size가 3이고, 앞의 3글자, 뒤의 3글자를 비교한 뒤 마지막에 2글자가 남은 상황에서 마지막 2글자는 압축할 수 없기 때문에 무조건 더해지게 되며 문자열 압축은 종료된다. 이러한 상황을 고려하기 위해 cmp r9, r8을 두어 (r9 = 남은 문자열 개수, r8 = chunk size)를 두어 back으로 보내지게 되는데

|  |
| --- |
| back:  add r12, r9 ; 남은 문자 개수  ret |

back 분기점은 남은 문자열의 개수를 모두 더해주고 반환한다.

**code\_3 – test case**









