Computer Networks and Security

88-447 Summer

Dr. H. Wu

Lab 02

Jason Choquette

104337378

Assigned Ciphertext

YFLWIMJRGYEXMAFQEEMQIMTRMFGYXPRRUCDJSEFCFQCSHPEXWVFFFSJZZPKGEIDLPMWUGCDXEYKQX
IRQDPRSVZNSFPMADBMKEVMQFXLRAPUKLGKGSLXGGCHMZVCQQRWRNDEPYZACDMRTEGDIXUZREIIZD
BFSJVMBQBTEDQEMSANLXCMASFAWIJNLPIVSTJFEAAXCKIWBEFQVWEDKMVONAJQICRRBUJJRQCZXJEN
KMRCBSFQVABLYZWXUZRULEIDCHIVXMMIRXUDGZXIARCBSARQMRWXVKJZIWFRFQTSFRCEWIQVFUGLA
DTQVXUDJQWWPNLHICRCRTIMZOPQWWVNLAJEJHJPYRGZKQHWCHPUXMAZLQBUHHQUXIYXAUZMYHQ
QHFBCWMPPGGCEIXUHLSWEEDZGVRGHLFSQLLCYSVLHQTEPYMCHIVSNPSIXGGCYWLRFPQIXRCKQAMG
GYRIAJNPPWSSOJQEWNMRIIPPNKQMRNKMIGPRZPHSMPDYZHMFZLWMRGNYNEWXDROLEVQDQIPVME
PMWGHLOXPLFJMHXUZRULEQZAOITGDBVSLARGZZMGZRUSRZQQOEZRMBUWLTZTQQIFNKQXINZLPLIEE
CIUYVDRDIQNQIELIVFFFIRRCKKJMERRUQTEDQEMSANDTIVNRYFLSENSSLPLEYEGMAZRURKJNKMREAZNB
VIPHYFMZRKGEXIADPUWEYVYKWWGHKGPEGHLSERQHBQWGEHZQHMAZFGQSENSEQEAMCDGIESYURM
ABGPIRGRMRQCPNLHEPRRAQRXUNKQMRNVYKALVBFUJPNSRQVQLRCXJKEDYFPCNLSEIHZXFAWXRRQVS
LANDOSYERCSSSQECXPSJSFAYKUGCUWGBTJPLEECJKFIPZJXIHNAPUPPVZLFGSAUCDWEGHMZEPVRRMXX
UZRYSQRMRMAIYKPQQIZACDIHINGOIJYNYFIHGGPAYKUSFQSTRMDDIRPGUURHBVLQEVNSFMRHGGCZCS
HKJIVMGDRAXLROPURGRRQMJXRQRQEEYEPQHMYKUDMXRSMXEHLSYPQMARRQVJBQRTIWRBMZHHNX
KKWIYEMDWLNKJIIANHRGRXVKUQLINQDDSQGGCBVMABCEWMABYEISSZPQJYFZJXEHLSYPQMARRQVQ
VFFFSTRMGFXLREGDWXQZWMRHZQQOVSFAGQXLRRCOSRQSFQRXUDPQWXUDBGGLRRQMFSHSRTIWP
GMAPJRSCFLIEDUMWXUDKGVQHQMREQNMQHSMPDYZHXUDLYVWVMEXIXUNPBWVBRCURVROJKCIFB
CDXEVMJKEJGDPFIEJHJXHSDTGFIARKJKSYNQCESXUNSSLXSTJMPJEDBPIEESFQJVRMATAMACMIWAHMEA
TIAZJUXXYDUUHIEZLPELNMBESQRVFUXIUZGDIHBKBXEHLVGFLEFNKQALNSKMWXRQDGPGNRRAJJRZRGV
IFRRQTTRCMGXSSHRARXBSFQPEJMYYERSNJXSARCFQVEFTESIWGHMZSJQDDQVIABCURLVRKMRRRQKD
WMAFJQXLBQNSVIRSCPQIJHRTIJSTQUSRJGWUJMGHQZXXBNBQPMTGRRYPGNQQICBTYSEMALPTEWGHL
SWESSCDEPYSFQWILDYDWEYEPQHHNQJURKZQFMWXVMEEQCUTQNERQHJASORCUUXLFNKQGYEHME
MXLZRMPJEDBPEVYHLSLIPDPFEMAKWEXVHBIMVEGGCDEPVDLZSXRHBUHRBSUARHRQYFNSUMMNNIPS
GZKXBGGEFINQBUXANRMZISSSFQPSAFCEXEACZXEGXDQFMLNUCQZIERCQRLRVMDIKBKBDMQZDBBMRP
DLQDEACFMHEPTPUSYFHKBEWFHTUXCBEDQEXHQCUXWGQSOOQRSFMXLRLGSLXYNMWREGTPMPSAZ
QFEKRASFAEFRRDERTDJKSYGNDBPEPDGZVINKJUJIUHQHSMPDUMWVNSFQVHRDNMRHHMAFYSHRFQTP
NBCPEABNBQRLNMBURQV

Encryption Key

Could not find

Plaintext

N/A – Could not decrypt

The number of coincidences N(n) as a function of the number of shifts n:

N(1) = 96
N(2) = 74
N(3) = 64
N(4) = 85
N(5) = 96
N(6) = 119
N(7) = 98
N(8) = 75
N(9) = 81
N(10) = 76
N(11) = 78
N(12) = 137
N(13) = 94
N(14) = 83
N(15) = 71
N(16) = 81
N(17) = 79
N(18) = 131
N(19) = 89
N(20) = 88

Very likely the key length is 6.

Inner products:

```
W[1]    =  0.007202
W[2]    =  0.005538
W[3]    =  0.006313
W[4]    =  0.006686
W[5]    =  0.006013
W[6]    =  0.005535
W[7]    =  0.006348
W[8]    =  0.007845
W[9]    =  0.006372
W[10]   =  0.006535
W[11]   =  0.006854
W[12]   =  0.007236
W[13]   =  0.005738
W[14]   =  0.005317
W[15]   =  0.005976
W[16]   =  0.006708
W[17]   =  0.005791
W[18]   =  0.006136
W[19]   =  0.007511
W[20]   =  0.005771
W[21]   =  0.005229
W[22]   =  0.006798
W[23]   =  0.010601
W[24]   =  0.006400
W[25]   =  0.005081
W[26]   =  0.005513
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <ctype.h>
#include "stringBuilder.h"


#define ALPHABET_LENGTH 26
double A[ALPHABET_LENGTH]; // letter frequencies
int ciphertext_length;


int Mod(int a, int b)
{
    return (a % b + b) % b;
}

int GCD(int x, int y)
{
    if (y == 0) return x;
    return GCD(y, x % y);
}

char* Cipher(char* input, char* key, bool encipher)
{
    int keyLen = strlen(key);

    for (int i = 0; i < keyLen; ++i)
        if (!isalpha(key[i]))
            return ""; // Error

    int inputLen = strlen(input);
    char* output = (char*)malloc(inputLen + 1);
    int nonAlphaCharCount = 0;

    for (int i = 0; i < inputLen; ++i)
    {
        if (isalpha(input[i]))
        {
            bool cIsUpper = isupper(input[i]);
            char offset = cIsUpper ? 'A' : 'a';
            int keyIndex = (i - nonAlphaCharCount) % keyLen;
            int k = (cIsUpper ? toupper(key[keyIndex]) : tolower(key[keyIndex])) - offset;
            k = encipher ? k : -k;
            char ch = (char)((Mod(((input[i] + k) - offset), 26)) + offset);
            output[i] = ch;
        }
        else
        {
            output[i] = input[i];
            ++nonAlphaCharCount;
        }
    }

    output[inputLen] = '\0';
    return output;
}

char* Encipher(char* input, char* key)
{
    return Cipher(input, key, true);
}

char* Decipher(char* input, char* key)
{
    return Cipher(input, key, false);
}






int GetKeyLength(int ioc[2000])
{
    int max1 = 0;
    int max2 = 0;
    int index1 = 0;
    int index2 = 0;

    for (int i = 0; i < 2000; i++)
    {
        if (ioc[i] > max1 && ioc[i] > max2)
        {
            max2   = max1;
            index2 = index1;
            max1   = ioc[i];
            index1 = i;
        }
    }

    return GCD(index1, index2);
}


/******************************************************************************
FUNCTION        : encrypt

DESCRIPTION     : This function encrypts a character based on a key as the
```

```c
parameter.

INPUT           : Type          : char
: Description   : The character to encrypt.

: Type          : int
: Description   : The shift key.

OUTPUT          : Type          : char
: Description   : The key-shifted chracter
*******************************************************************************/
char encrypt(char ch, int key)
{
    if (!isalpha(ch)) return ch;
    char offset = isupper(ch) ? 'A' : 'a';
    return (char)((((ch + key) - offset) % 26) + offset); // shift cipher
}


/*******************************************************************************
FUNCTION        : decrypt

DESCRIPTION     : This function decrypts each character of the ciphertext using the
                  given key parameter.

INPUT           : Type          : char *
                : Description   : The ciphertext.

                : Type          : int
                : Description   : The shift key.

OUTPUT          : Type          : char *
                : Description   : The decrypted text
*******************************************************************************/
char * decrypt(char * text, int key)
{
    int text_length = strlen(text);
    char * plaintext = (char*)malloc(text_length + 1);

    for (int i = 0; i < text_length; i++)
        plaintext[i] = encrypt(text[i], key);

    plaintext[text_length] = '\0'; // add null termination character

    return plaintext;
}




/*******************************************************************************
```

```c
FUNCTION        : innerProduct

DESCRIPTION     : This function uses the frequencies of letters expected
in an english message that has been Caeser-shifted i
letters to the left by a 26-dimensional vector, A.

One of these vectors should agree fairly closely with the
frequencies of letters we see in our ciphertext.
Which vector that is tells us the shift amount for our sampling,
and the first letter of our keyword.

To find the vector in the previous list above that most closely
matches the vector u, we recall that the dot product of two
vectors is connected to the angle θ between those two vectors
in the following way:

W.A=|W||A|cosθ

If we want to find the two vectors W and Ai that most closely
match, we want to find the two vectors with the smallest
angle between them.

Noting that smaller angles produce larger cosine values and
also noting that the magnitude of the denominator is the same
for every vi as the same 26 numbers are involved each time
(just in different orders), we can simply seek the two vectors W
and Ai whose dot product is largest.


INPUT           : Type          : int[]
: Description   : The letter frequencies of the ciphertext.

: Type          : int *
: Description   : A reference to the encryption key.

OUTPUT          : Type          : double *
: Description   : The array of innerproducts.
******************************************************************************/
double * innerProduct(int W[], int * key)
{
    double inner_product[ALPHABET_LENGTH] = { 0 };
    double sum = 0;
    int j;


    for (int i = 0; i < ALPHABET_LENGTH; i++)
    {
        for (j = 0; j < ALPHABET_LENGTH; j++)
            sum += W[j] * A[(j + i) % ALPHABET_LENGTH]; // shift the frequency array

        inner_product[i] = sum / ciphertext_length;

        // find the largest innerproduct. This will be the key.
        if (inner_product[*key] < inner_product[i]) *key = i;

        // reset counter and sum
        j = 0;
        sum = 0;
    }

    printf("Inner products: \n\n");
    for (size_t i = 0; i < 26; i++)
        printf("W[%d]\t=  %f\n", i + 1, inner_product[i]);

    return inner_product;
}



int main()
{

#pragma region Completed
    // english alphabet letter frequncies
    A[0] = 0.08167;
    A[1] = 0.01492;
    A[2] = 0.02782;
    A[3] = 0.04253;
    A[4] = 0.12702;
    A[5] = 0.02228;
    A[6] = 0.02015;
    A[7] = 0.06094;
    A[8] = 0.06996;
    A[9] = 0.00153;
    A[10] = 0.00772;
    A[11] = 0.04025;
    A[12] = 0.02406;
    A[13] = 0.06749;
    A[14] = 0.07507;
    A[15] = 0.01929;
    A[16] = 0.00095;
    A[17] = 0.05987;
    A[18] = 0.06327;
    A[19] = 0.09056;
    A[20] = 0.02758;
    A[21] = 0.00978;
    A[22] = 0.02360;
    A[23] = 0.00150;
    A[24] = 0.01974;
    A[25] = 0.00074;

    char * ct =
      "YFLWIMJRGYEXMAFQEEMQIMTRMFGYXPRRUCDJSEFCFQCSHPEXWVFFFSJZZPKGEIDLPMWUGCDXEY
```

```
     KQXIRQDPRSVZNSFPMADBMKEVMQFXLRAPUKLGKGSLXGGCHMZVCQQRWRNDEPYZACDMRTEGDIXUZRE
     IIZDBFSJVMBQBTEDQEMSANLXCMASFAWIJNLPIVSTJFEAAXCKIWBEFQVWEDKMVONAJQICRRBUJJR
     QCZXJENKMRCBSFQVABLYZWXUZRULEIDCHIVXMMIRXUDGZXIARCBSARQMRWXVKJZIWFRFQTSFRCE
     WIQVFUGLADTQVXUDJQWWPNLHICRCRTIMZOPQWWVNLAJEJHJPYRGZKQHWCHPUXMAZLQBUHHQUXIY
     XAUZMYHQQHFBCWMPPGGCEIXUHLSWEEDZGVRGHLFSQLLCYSVLHQTEPYMCHIVSNPSIXGGCYWLRFPQ
     IXRCKQAMGGYRIAJNPPWSSOJQEWNMRIIPPNKQMRNKMIGPRZPHSMPDYZHMFZLWMRGNYNEWXDROLEV
     QDQIPVMEPMWGHLOXPLFJMHXUZRULEQZAOITGDBVSLARGZZMGZRUSRZQQOEZRMBUWLTZTQQIFNKQ
     XINZLPLIEECIUYVDRDIQNQIELIVFFFIRRCKKJMERRUQTEDQEMSANDTIVNRYFLSENSSLPLEYEGMA
     ZRURKJNKMREAZNBVIPHYFMZRKGEXIADPUWEYVYKWWGHKGPEGHLSERQHBQWGEHZQHMAZFGQSENSE
     QEAMCDGIESYURMABGPIRGRMRQCPNLHEPRRAQRXUNKQMRNVYKALVBFUJPNSRQVQLRCXJKEDYFPCN
     LSEIHZXFAWXRRQVSLANDOSYERCSSSQECXPSJSFAYKUGCUWGBTJPLEECJKFIPZJXIHNAPUPPVZLF
     GSAUCDWEGHMZEPVRRMXXUZRYSQRMRMAIYKPQQIZACDIHINGOIJYNYFIHGGPAYKUSFQSTRMDDIRP
     GUURHBVLQEVNSFMRHGGCZCSHKJIVMGDRAXLROPURGRRQMJXRQRQEEYEPQHMYKUDMXRSMXEHLSYP
     QMARRQVJBQRTIWRBMZHHNXKKWIYEMDWLNKJIIANHRGRXVKUQLINQDDSQGGCBVMABCEWMABYEISS
     ZPQJYFZJXEHLSYPQMARRQVQVFFFSTRMGFXLREGDWXQZWMRHZQQOVSFAGQXLRRCOSRQSFQRXUDPQ
     WXUDBGGLRRQMFSHSRTIWPGMAPJRSCFLIEDUMWXUDKGVQHQMREQNMQHSMPDYZHXUDLYVWVMEXIXU
     NPBWVBRCURVROJKCIFBCDXEVMJKEJGDPFIEJHJXHSDTGFIARKJKSYNQCESXUNSSLXSTJMPJEDBP
     IEESFQJVRMATAMACMIWAHMEATIAZJUXXYDUUHIEZLPELNMBESQRVFUXIUZGDIHBKBXEHLVGFLEF
     NKQALNSKMWXRQDGPGNRRAJJRZRGVIFRRQTTRCMGXSSHRARXBSFQPEJMYYERSNJXSARCFQVEFTES
     IWGHMZSJQDDQVIABCURLVRKMRRRQKDWMAFJQXLBQNSVIRSCPQIJHRTIJSTQUSRJGWUJMGHQZXXB
     NBQPMTGRRYPGNQQICBTYSEMALPTEWGHLSWESSCDEPYSFQWILDYDWEYEPQHHNQJURKZQFMWXVMEE
     QCUTQNERQHJASORCUUXLFNKQGYEHMEMXLZRMPJEDBPEVYHLSLIPDPFEMAKWEXVHBIMVEGGCDEPV
     DLZSXRHBUHRBSUARHRQYFNSUMMNNIPSGZKXBGGEFINQBUXANRMZISSSFQPSAFCEXEACZXEGXDQF
     MLNUCQZIERCQRLRVMDIKBKBDMQZDBBMRPDLQDEACFMHEPTPUSYFHKBEWFHTUXCBEDQEXHQCUXWG
     QSOOQRSFMXLRLGSLXYNMWREGTPMPSAZQFEKRASFAEFRRDERTDJKSYGNDBPEPDGZVINKJUJIUHQH
     SMPDUMWVNSFQVHRDNMRHHMAFYSHRFQTPNBCPEABNBQRLNMBURQV";
    //char * ct =
     "DWGFQRVPLNYIEQBYUOXFVHPGXFIGXNTKZSNRKKXSTUTLKLDAENUPDULVSIUMAVRGEFKYEQXCCV
     NOBZWJSOHTPVGKPEFRRKZFIBCOHCFRLRHPSZECDIUYZIAVIVSENGHTTIOHAXNGSEIGLHGBQDVSY
     QERIASTJSEINHKGNBOVGOVSZLVKENIOUEXOUSFYVCTJOIAFIERKBEEQAUUQTLDUTOOOTCIUOFHR
     WAPNIRVIIPQKEFXTKCYRFXNIVQTUDRRCNUGLHCDPORHIVWQAAEOKBATFWRWQSEQWIUCTOHADGBE
     IPPNPYFSNNBWDUTVHSWQSEFIIXOMWVADKNQASAAURQDNRRQCEMRLAUSFPBHSKLXEGWAVWDSVCGN
     OFHBGPUWUNQLAUNQRNCGGNTAQHHGCAMRUAPDMSGXCKNQABUDGWANVPCCVBOFHEUCUOAPNFSRTUP
     TYODEFDWCCUTADTCVEOCDSUSNLRIHCDEHRBIIRFHNKEVKWEAWETYINYXFGSIAFPBQEFTBTXRYGN
     QIHGCQTUTOTSQSGDPQSDOGLHGXTIFDWPGARQHDKCFRNRTGNYEPDMGRQSNXDPYITBTXCWUNRIHGM
     AFSTEEEBSZNDGKDPBXRQDIHNIOPOMRGWIUDTETDOFYRTUPTPYITUPTYOWNBLADYGTGWEEYOOBWL
     CVMTUPTOSEEEPBNOOOPDCTSQDCDITYFFYXPRKZTYNHGVMUTWEFGUTUPPRKDEAIEPTAYZTNVBMIF
     XNIRUSNGMUDAHRPVGXUNZDCMNQSCPITSZWUPTKMAUYSNQDNUGROPCUDRGTJOIOEHTRYESVQLGDM
     SGTAPNMNLLAASEAVSWKDTIARRGKEIAVCQVPNRHSCCYRFXNIVQTUDRRDAOXWETMAFSTEWZETNXRU
     GUTUWETSPOADTUOQWUPTAYGEKEEEDFOSXNFEZLRHSAYGCBCSKNQRVILKUQLLIHCDIEFWANVPIFR
     OXODACPCMOFOSHTTIOHAXNGYZTUTCQPRERIRCIBOVGOVGMSFDBGBQDNIOPMQCBBEEYYEZNFTSQN
     QWEUKUDFAIRZUNTWIUKDMFIHTYGGUBIPOZEIDUUPMCUTZRKEAYAOYWQTBXNVODEFIMACQLSXNOI
     OOSUEGMGPFPNFSIIYARGCBEPIYQEDCBROVRQRRXSKDMBNGGCSZHRLAUCAQHPIPDXYUJMQBAUFIH
     CDUWNHFQBOEQIONKGGUPNFGQWRCTVYSEGWETDATUTDTKIIAVRQYYWUTRGDTEPDFHOQCHESCXPTE
     PYTOYAVCEFEZDVHTWBNEQPSYOTAQAEHDFHRBPQSDOGBAFOYEETCCZUTHAAVOFHRHCGXQOSIHGXU
     GUIBGPARRAIUDQNVCGXODYPPRGPGLYNAPNHEEXFASZGGWERYEIGXOPYRTUTVCBUOHHCWZESBBRU
     MMVRCDKCTSGDOFLKTUTTTKKAASPQEDEQDUVIQSGWEPCTEPPMGKORBHSVYFHRLIPNAWJWETOKOHH
     AVGUTUBAFOYOVHENVQCLCTJSMYRHHGBQAETTJOFHETEEEBSNCDVRQCHEOPDTEZPNVOXPVTCGRML
     SSRWXWTUPTYYGLQQEOBXAJGEPMQCNKEPNUSUHAPNFHRDNGYZTUTTTKKJBWNEKHEASIUREIFPWJS
     YPHIIVNAWAIHGBQGBDDQXQTJDTJBQESDUTPUVRQUVGTEETTJOZIFIHGMGPBUMTSZGYTTJYDPUTD
     QOENBITCUQCBUFGOFHRCANVMRRPCEYGNGTDHYDOATMQWQNGBYHBUEASWKDTIAUIPSFEPPRGRQTB
     DKCNDOCDRVGAFEDMVRQGEDUPNEIATAEROUCHECVUNTIHGWGPVCSGZMRNIEVOETGJBGCFAFIIPQQ
```

```
        APWIPDGRAPSJOPIQHOJSEPUNSKYSNBBYWXPEELEPDMCHGIQEECUPNIOMNRMPTOESVDNIKFHRGEFDTEETTJKFIPPNQXXYQTSEBUBRPSJKXFCJZBVQDNCDJKXFETLKOH";

    int shift = 1;
    int indexPos = 0;
    ciphertext_length = strlen(ct);

    // Kerckhoff's Method Part A:
    int IOC[2000] = { 0 }; // Incedents of Coincidence

    for (int i = shift; i < ciphertext_length - 1; i++)
    {
        for (int j = i; j < ciphertext_length; j++)
        {
            if (ct[j] == ct[indexPos]) IOC[shift] += 1;
            indexPos++;
        }

        shift++;
        indexPos = 0;
    }


    printf("%s\n\n", ct);
    printf("Number of letters in the text : %d\n\n", ciphertext_length);
    printf("The number of coincidences N(n) as a function of the number of shifts n :\n\n");

    for (int i = 1; i < 21; i++)
        printf("N(%d) = %d\n", i, IOC[i]);

    int keyLength = GetKeyLength(IOC);
    printf("\n\nVery likely the key length is %d.\n", keyLength);

    // GOOD UP TO HERE! //

#pragma endregion


    // Kerckhoff's Method Part B:
    /*
        1. Assume the key length is l.
        2. Split the ciphertext into l parts.
        3. For i = 1 to l
            a) Treat part i as the ciphertext resulted form a shift cipher.
            b) Decrypt part i using the method for cracking a shift cipher.
            c) If i = l, break; Else i++ and go back to step 3.
    */

    int j;
    for (int i = 0; i < keyLength; i++)
    {
        j = i;
        stringbuilder* ciphertext = sb_new();
        for (; j < ciphertext_length; j+= keyLength)
            sb_append_ch(ciphertext, ct[j]);

        sb_make_cstring(ciphertext);
        //printf("%s\n", sb_make_cstring(ciphertext));

        int ct_length = strlen(sb_make_cstring(ciphertext));
        int W[ALPHABET_LENGTH] = { 0 };
        int key = 0;

        // count occurences
        for (int i = 0; i < ct_length; i++)
            W[(int)sb_make_cstring(ciphertext)[i] - 65] += 1;

        // compute inner product W . Ai, and find the key
        double * inner_product = innerProduct(W, &key);

        // get each kth letter and form a new array....
        char * pt = decrypt(sb_make_cstring(ciphertext), key);

        printf("key %d : %d\n", i ,key);
    }


    //printf("A likely encryption key can be obtained as : key = %s", key);
    //printf("The plaintext recovered with the above key is as follows : \n%s", pt);

    getchar();

    return 0;
}
```

```c
/**
* Stringbuilder - a library for working with C strings that can grow dynamically as they are appended
*
*/

#include <stdlib.h>
#include <string.h>
#include "stringbuilder.h"


/**
* Creates a new stringbuilder with the default chunk size
*
*/
stringbuilder* sb_new()
{
    return sb_new_with_size(1024);
}

/**
* Creates a new stringbuilder with initial size at least the given size
*/
stringbuilder* sb_new_with_size(int size)
{
    stringbuilder* sb;

    sb = (stringbuilder*)malloc(sizeof(stringbuilder));
    sb->size = size;
    sb->cstr = (char*)malloc(size);
    sb->pos = 0;
    sb->reallocs = 0;

    // Fill cstr with null to ensure it is always null terminated
    memset(sb->cstr, '\0', size);

    return sb;
}

void sb_reset(stringbuilder* sb)
{
    sb->pos = 0;
    memset(sb->cstr, '\0', sb->size);
}

/**
* Destroys the given stringbuilder
*/
void sb_destroy(stringbuilder* sb, int free_string)
{
    if (free_string)
        free(sb->cstr);

    free(sb);
}

/**
* Internal function to resize our string buffer's storage.
* \return 1 iff sb->cstr was successfully resized, otherwise 0
*/
int sb_resize(stringbuilder* sb, const int new_size)
{
    char* old_cstr = sb->cstr;

    sb->cstr = (char *)realloc(sb->cstr, new_size);

    if (sb->cstr == NULL)
    {
        sb->cstr = old_cstr;
        return 0;
    }

    memset(sb->cstr + sb->pos, '\0', new_size - sb->pos);
    sb->size = new_size;
    sb->reallocs++;
    return 1;
}

int sb_double_size(stringbuilder* sb)
{
    return sb_resize(sb, sb->size * 2);
}

void sb_append_ch(stringbuilder* sb, const char ch)
{
    int new_size;

    if (sb->pos == sb->size)
        sb_double_size(sb);

    sb->cstr[sb->pos++] = ch;
}

/**
* Appends at most length of the given src string to the string buffer
*/
void sb_append_strn(stringbuilder* sb, const char* src, int length)
{
    int chars_remaining;
    int chars_required;
    int new_size;

    // <buffer size> - <zero based index of next char to write> - <space for null terminator>
    chars_remaining = sb->size - sb->pos - 1;
    if (chars_remaining < length)
    {
        chars_required = length - chars_remaining;
        new_size = sb->size;
        do {
            new_size = new_size * 2;
        } while (new_size < (sb->size + chars_required));

        sb_resize(sb, new_size);
    }

    memcpy(sb->cstr + sb->pos, src, length);
    sb->pos += length;
}

/**
* Appends the given src string to the string builder
*/
void sb_append_str(stringbuilder* sb, const char* src)
{
    sb_append_strn(sb, src, strlen(src));
}


/**
* Allocates and copies a new cstring based on the current stringbuilder contents
*/
char* sb_make_cstring(stringbuilder* sb)
{
    if (!sb->pos)
        return 0;

    char* out = (char*)malloc(sb->pos + 1);
    strcpy(out, sb_cstring(sb));

    return out;
}
```

```c
#ifndef STRINGBUILDER_H
#define STRINGBUILDER_H

typedef struct stringbuilder_tag {
    char* cstr;             /* Must be first member in the struct! */
    int   pos;
    int   size;
    int   reallocs;         /* Performance metric to record the number of string reallocations */
} stringbuilder;

/**
* Creates a new stringbuilder with the default chunk size
*
*/
stringbuilder* sb_new();

/**
* Destroys the given stringbuilder.  Pass 1 to free_string if the underlying c string should also be freed
*/
void sb_destroy(stringbuilder* sb, int free_string);

/**
* Creates a new stringbuilder with initial size at least the given size
*/
stringbuilder* sb_new_with_size(int size);

/**
* Resets the stringbuilder to empty
*/
void sb_reset(stringbuilder* sb);

/**
* Appends the given character to the string builder
*/
void sb_append_ch(stringbuilder* sb, const char ch);

/**
* Appends at most length of the given src string to the string buffer
*/
void sb_append_strn(stringbuilder* sb, const char* src, int length);

/**
* Appends the given src string to the string builder
*/
void sb_append_str(stringbuilder* sb, const char* src);

/**
* Allocates and copies a new cstring based on the current stringbuilder contents
*/
char* sb_make_cstring(stringbuilder* sb);
```

```

/**
* Returns the stringbuilder as a regular C String
*/
#define sb_cstring(sb) ((sb)->cstr)

#endif // STRINGBUILDER_H

```