1.

```
#include <stdio.h>
#include <string.h>

int main ()
{
    int n_pedras, n_sapos;
    int i, j, pos_sapo, dist_sapo;
    int pedras [100];

    // Lê os 2 primeiros valores.
    scanf ("%d %d", &n_pedras, &n_sapos);

    /* A "chave" para resolver o problema é o vetor pedras. Iniciamos ele com 0 em
       todas as posições, indicando que nenhuma tem sapo. Depois, para cada sapo,
       percorremos o vetor indicando posições este um sapo pode ter parado. */
    for (i = 0; i < n_pedras; i++)
        pedras [i] = 0;

    // Para cada sapo...
    for (i = 0; i < n_sapos; i++)
    {
        // Lê os 2 valores.
        scanf ("%d %d", &pos_sapo, &dist_sapo);

        // Indica que a posição inicial pode ter um sapo.
        pedras [pos_sapo] = 1;

        // Faz o sapo saltar para a esquerda!
        for (j = pos_sapo - dist_sapo; j >= 0; j -= dist_sapo)
            pedras [j] = 1;

        // Faz o sapo saltar para a direita!
        for (j = pos_sapo + dist_sapo; j < n_pedras; j += dist_sapo)
            pedras [j] = 1;
    }

    // Mostra.
    for (i = 0; i < n_pedras; i++)
        printf ("%d", pedras [i]);

    return 0;
}
```

2. Vou mostrar 3 versões, mas todas usam esta função:

```
int ehLetra (char c) {
    return ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'));
}

/* Versão 1: Bem simples! Procura por letras seguidas de não-letras. */
int contaPalavras (char* string) {
    int i;
    int n_palavras = 0;

    for (i = 0; string [i] != '\0'; i++)
        if (ehLetra (string [i]) && !ehLetra (string [i+1]))
            n_palavras++;

   return (n_palavras);
}

/* Versão 2: Mais complicada. Se acha uma letra, vê se já estava em uma palavra.
   A ideia é contar cada palavra INICIADA. */
int contaPalavras (char* string) {
    int i;
    int n_palavras = 0;
    int palavra = 0;

    for (i = 0; string [i] != '\0'; i++)
    {
        if (ehLetra (string [i]))
        {
            if (!palavra)
            {
                n_palavras++;
                palavra = 1;
            }
        }
        else
            palavra = 0;
    }

   return (n_palavras);
}

/* Versão 3: Mais complicada, quando encontra uma letra tenta percorrer a
   palavra até o final. */
int contaPalavras3 (char* string) {
    int i = 0, j;
    int n_palavras = 0;

    while (string [i] != '\0')
    {
        j = 1;
        if (ehLetra (string [i]))
        {
            while (ehLetra (string [i+j]))
                j++;
            n_palavras++;
        }

        i += j;
    }

    return (n_palavras);
}
```

3.

```
/* Função auxiliar de busca. */
int presenteEm (int valor, int* v, int n_elementos)
{
    int i;
    for (i = 0; i < n_elementos; i++)
        if (v [i] == valor)
            return (1);

    return (0);
}

/* Coloca em v_out os elementos de v_in, mas sem repetições. */
int removeRepeticoes (int* v_in, int n_elementos, int* v_out)
{
    int i;
    int n_out = 0;

    /* Usamos uma função de busca auxiliar! Só guarda um elemento se ele ainda
       não tinha aparecido no vetor de saída. */
    for (i = 0; i < n_elementos; i++)
        if (!presenteEm (v_in [i], v_out, n_out))
            v_out [n_out++] = v_in [i];

    return (n_out);
}
```

4.

```
/* Começamos marcando onde começou a sequência não-decrescente atual. O loop
  mais externo não percorre o vetor posição-a-posição - em vez disso, ele pula
  blocos inteiros, até que tenhamos uma sequência que começa após o final do
  vetor. O loop mais interno é que percorre o vetor posição-a-posição, começando
  onde a última sequência havia terminado e indo até que o vetor ou a sequência
  termine (i.e. até encontrarmos um valor maior que o anterior). Neste momento,
  verificamos se a última sequência é maior do que a maior que já  havíamos
  encontrado. */
int tamMaiorSeqNDec (int* val, int n, int* inicio, int* fim)
{
    int inicio_atual = 0, i, tam_maior = 0;

    /* Para cada sequência... */
    while (inicio_atual < n)
    {
        /* Percorre o vetor até o final desta sequência. */
        i = inicio_atual+1;
        while (i < n && val [i] >= val [i-1])
            i++;

        /* Esta sequência é a maior que já vimos? */
        if (i-inicio_atual > tam_maior)
        {
            tam_maior = i-inicio_atual;
            *inicio = inicio_atual;
            *fim = i-1;
        }

        inicio_atual = i; /* A próxima sequência começa onde esta terminou. */
    }

    return (tam_maior);
}
```

5. Vou colocar aqui 3 versões!

```
/* Versão 1 */

#include <stdio.h>

#define MAX_N 1024

int main ()
{
    int i, n_salas;
    int vidas [MAX_N]; // Uma posição para cada sala.
    int entrada, saida, n_vidas, max_vidas;

    // Começa lendo o número de salas e a quantidade de vidas em cada sala.
    scanf ("%d", &n_salas);
    for (i = 0; i < n_salas; i++)
        scanf ("%d", &(vidas [i]));

    /* O jeito simples de resolver este problema é testar, para cada ponto de
       entrada, todas as saídas possíveis. Estou supondo que existe pelo menos
       uma sala i tal que vidas [i] > 0. Note que eu usei aqui variáveis com
       nomes de entrada e saída para controlar os loops. Poderia ser i e j, mas
       neste contexto pode ser útil usar nomes um pouco mais longos. */

    max_vidas = 0;
    for (entrada = 0; entrada < n_salas; entrada++)
    {
        // Indo para a esquerda (ou saindo pela entrada!)...
        n_vidas = 0;
        for (saida = entrada; saida >= 0; saida--)
        {
            n_vidas += vidas [saida];
            if (n_vidas > max_vidas)
                max_vidas = n_vidas;
        }

        /* Indo para a direita. Note que o miolo deste loop é o mesmo do outro
           Daria para usar um único loop, você consegue imaginar como? Funciona,
           mas na verdade fica um pouco feio... */
        n_vidas = vidas [entrada];
        for (saida = entrada+1; saida < n_salas; saida++)
        {
            n_vidas += vidas [saida];
            if (n_vidas > max_vidas)
                max_vidas = n_vidas;
        }
    }

    printf ("%d\n", max_vidas);

    return 0;
}
```

```
/* Versão 2: embora a primeira solução funcione, ela pensa em termos de entradas
e saídas. Mas note que isso não era necessário! Entrar pela sala A e sair pela B
dá o mesmo resultado que entrar pela B e sair pela A. E para este problema, só
importa o número de vidas! Então, uma solução mais eficiente evitaria trechos
que já apareceram. Dá para só ir da esquerda para a direita! O começo e o
final são iguais, mas o loop central pode ser mais simples. */

    max_vidas = 0;
    for (entrada = 0; entrada < n_salas; entrada++)
    {
        n_vidas = 0;
        for (saida = entrada; saida < n_salas; saida++)
        {
            n_vidas += vidas [saida];
            if (n_vidas > max_vidas)
                max_vidas = n_vidas;
        }
    }


/* Versão 3: parece que ficou bom, né? Mas dá para melhorar ainda mais. Ainda
 suponha que estamos percorrendo da esquerda para a direita. O que realmente
 importa é o maior número de vidas que podemos obter SAINDO por uma porta.
 A entrada não importa tanto, o que importa é se indo por algum caminho, o
 número de vidas em algum momento se torna negativo. Se isso acontecer, é porque
 tem uma sala pela qual não vale a pena passar. Assim, a entrada fica implícita
 como sendo a primeira sala, se o número de vidas ficar negativo, movemos a
 entrada (implicitamente!) para a próxima sala e começamos a contar de novo! Não
 é tão intutivo assim, mas o loop central seria só um for! */

    max_vidas = 0;
    n_vidas = 0;
    for (saida = 0; saida < n_salas; saida++)
    {
        n_vidas += vidas [saida];

        if (n_vidas > max_vidas)
            max_vidas = n_vidas;

        if (n_vidas < 0) // Não vale a pena passar por aqui!
            n_vidas = 0;
    }
```