```
int particionar (int *v, int ii, int is) {
  int esq=ii, dir=is, pivo=v[ii];
  while (esq<dir) {
    while (v[esq]<=pivo && esq<is)
      esq++;
    while (v[dir]>pivo)
      dir--;
    if (esq<dir) {
      int temp;
      temp = v[esq];
      v[esq]=v[dir];
      v[dir]=temp; } }
  v[ii]=v[dir];
  v[dir]=pivo;
  return dir; }
```

Classificação por Particionamento - quicksort

Exercício:

Agora, construa uma função recursiva, em C, que recebe um vetor de inteiros e o número de elementos neste vetor. Esta função deve ordenar o vetor implementando o quicksort.

```
void quicksort (int *v, int n)
{
  if (n>1)
  {
    int pont_part=particionar(v, 0, n-1);
    quicksort (v, pont_part);
    quicksort (&v[pont_part+1],
     n-1-pont_part);
  }
}
```

Classificação por Particionamento - quicksort

A tarefa de escolher o pivô pode ser executada de forma mais eficiente. A chave ideal para o pivô seria a mediana das chaves, com a qual se teria uma divisão a mais balanceada possível. Só que para determinar a mediana, de forma eficiente, é preciso ordenar o vetor! Como a distribuição das chaves é, em princípio, aleatória, qualquer uma tem a mesma chance de ser mediana. Mas, ao se tomar a primeira, como foi feito anteriormente, corre-se o risco de ser esta a menor (ou a maior) de todas as chaves, tornando praticamente inócuo o trabalho na primeira fase de particionamento (pois se teria uma partição sem nenhum elemento e outra com *n-1* elementos). Uma escolha melhor é a mediana entre a primeira chave, a última e a do meio do vetor.

Classificação por Particionamento - quicksort

Como um exercício, reescreva os algoritmos anteriores, considerando que o pivô não é mais o primeiro elemento mas, sim a mediana entre o primeiro elemento, o último e o elemento do meio do vetor.

Classificação por Particionamento - quicksort

O desempenho do algoritmo *quicksort* pode ser aquilatado com base nas considerações que seguem. Na fase de particionamento, todos os elementos são comparados com o pivô. Na pior hipótese, todas as chaves seriam trocadas (caso do vetor invertido). Logo, este é o processo **O**(n). Por outro lado, as partições vão diminuindo, e, na melhor hipótese, vão se subdivindo em duas partições de mesmo tamanho (tal ocorre naturalmente quando o vetor está ordenado ou invertido). Nesse caso, gasta-se $\log_2 n$ reclassificações até se chegar às partições de tamanho 1. O melhor desempenho deste processo é então de ordem n log n.

Classificação por Particionamento - quicksort

O pior caso ocorre quando o pivô escolhido é uma chave mínima (ou máxima), pois acarreta uma partição nula e outra de n – 1 elementos. Se isto se repetir em todas as reclassificações, serão necessárias n subsivisões até a conclusão, e o desempenho para o pior caso é **O**(n^2).

A chance de isso ocorrer, na versão melhora do processo que implementamos, é de apenas $1/n^3$ (se houver três chaves mínimas (ou máxima), ocupando exatamente a primeira posição, a intermediária e a última em cada partição).

Método dos Incrementos Decrescentes

Estudamos anteriormente o método de classificação/ordenação por inserção e vimos que o mesmo pode ser otimizado utilizando-se a busca binária ou implementando o mesmo sobre uma lista encadeada.

Porém, veremos agora como obter uma otimização mais significativa, o que denominaremos de ***classificação de incremento decrescente*** (ou ***Shell sort***), assim denominada em homenagem a seu descobridor Donald Shell.

Método dos Incrementos Decrescentes

Esse método classifica subvetores separados do vetor original. Esses subvetores contêm todo *k*ésimo elemento do vetor original. O valor de *k* é chamado ***incremento***.

Por exemplo, se *k* é 5, o subvetor consistindo em *v*[0], *v*[5], *v*[10],... é classificado primeiro. Cinco subvetores, cada um contendo um quinto dos elementos do arquivo original, são classificados dessa maneira. São eles:

Subvetor 0 -> *v*[0] *v*[5] *v*[10] ...

Subvetor 1 -> *v*[1] *v*[6] *v*[11] ...

Subvetor 2 -> *v*[2] *v*[7] *v*[12] ...

Subvetor 3 -> *v*[3] *v*[8] *v*[13] ...

Subvetor 4 -> *v*[4] *v*[9] *v*[14] ...

Método dos Incrementos Decrescentes

O *i*ésimo elemento do *j*ésimo subvetor é *v*[*i* * 5 + *j*]. Se um incremento *k* diferente for escolhido, os *k* subvetores serão divididos de modo que o *i*ésimo elemento do *j*ésimo subvetor seja *v*[*i* * *k* + *j*].

Depois que os primeiros *k* subvetores estiverem classificados (geralmente por inserção simples), será escolhido um novo valor menor que *k* e o vetor será novamente particionado em novos conjuntos de subvetores. Cada um desses subvetores será classificado e o processo se repetirá novamente com um valor ainda menor que *k*.

Método dos Incrementos Decrescentes

Em algum momento, o valor de *k* será definido como 1, de modo que o subvetor consistindo no vetor inteiro será classificado.

Uma seqüência decrescente de incrementos é determinada no início do processo inteiro. O último valor nessa seqüência deve ser 1.

Por exemplo, se o vetor original for:

75 25 95 87 64 59 86 40 16 49

e a seqüência (5, 2, 1) for escolhida, os seguintes subvetores serão classificados em cada iteração:

Método dos Incrementos Decrescentes

Vetor original: **75 25 95 87 64 59 86 40 16 49**

k	**vetor resultante**
5	59 25 40 16 49 75 86 95 87 64
2	40 16 49 25 59 64 86 75 87 95
1	**16 25 40 49 59 64 75 86 87 95**

Com base no que foi discutido, codifique uma função que receba um vetor (de inteiros) e o número de elementos no mesmo e através do método *shell sort* ordene de forma crescente os elementos do vetor.

OBS.: Inicialize o incremento com n/2 e faça-o decrescer nesta taxa.