Gabarito da Prova 2
`MAC338 - 20/6/2000`

Resposta da questão 1.

(i)

Sem o primeiro `for` não há garantia de que o `while` do código funciona corretamente, pois a condição `less(v, a[j-1])` pode ser verdadeira para $i = l + 2$ e $j = l + 1$, com conseqüências imprevisíveis. Por exemplo, tome `a[l..r]`={`2,3,1`} e `a[l-1]=0`.

(ii)

O primeiro `for` coloca o mínimo do vetor `a[]` em `a[l]`. Com isso, evita-se o problema descrito acima, pois o teste condicional no `while (less(v, a[j-1]))` eventualmente se tornará falso com $j \geq l + 1$, e não há necessidade de verificar se j é igual a $l + 1$, reduzindo assim o número de comparações. (Este é um bom exemplo de uso de *sentinelas*.)

Observação: Nos demais itens, o vetor `a[]` contém os números $\{0, \ldots, n - 1\}$.

(iii)

Para $n = 7$, $l = 0$ e $r = 6$ temos o vetor `a[] =` {`0, 3, 6, 2, 5, 1, 4`}. O número de inversões deste vetor é $0 + 2 + 4 + 1 + 2 + 0 + 0 = 9$.

(iv)

Suponha que em uma certa iteração do segundo `for` o elemento `a[i]` esteja envolvido em q inversões. A parte do vetor `a[]` de l a $i - 1$ já está ordenada. Então nesta iteração:

- a atribuição `a[j] = a[j-1]` é executada q vezes;
- a comparação `less(v, a[j-1])` é feita $q + 1$ vezes.

(v)

Seja j tal que $a[j] = i$. Sabemos que $|j - i| \leq m$.

Suponha primeiro que $i \geq j$. O número de elementos de `a[`j`]` a `a[`$i - 1$`]` maiores que i é no máximo m, pois $i - j \leq m$. O número de elementos de `a[0]` a `a[`$j - 1$`]` maiores que i é no máximo $m - 1$, pois senão um deles teria deslocamento maior do m. Logo, o número de inversões envolvendo i é no máximo $2m - 1$.

Se $j > i$ então o número de elementos de `a[0]` a `a[`$i - 1$`]` maiores que i é no máximo $m - 1$, pois senão um deles teria deslocamento maior do m. Nesse caso, o número de inversões envolvendo i é no máximo $m - 1$.

(vi)

Seja `a[]` uma instância onde o deslocamento de cada elemento do vetor é no máximo m.

O primeiro `for` gasta tempo $O(n)$. Em cada iteração i do segundo `for` o número de atribuções e comparações é proporcional ao número q de inversões nas quais $a[i]$ está envolvido pelo item (*iv*). Note que o programa nunca aumenta o número de inversões nas quais um elemento do vetor está envolvido. Pelo item (*v*), q é no máximo $2m - 1$. Logo, o segundo `for` gasta tempo $O(mn)$.

Portanto, o tempo de execução de `insertion(a,0,n-1)` é $O(n + mn) = O(mn)$.

Resposta da questão 2.

(i)

A parte *maior* é "empilhada" primeiro. Portanto, a parte *menor* é ordenada primeiro.

(ii)

Obviamente, $t(0) = t(1) = 1$ pois apenas o par $(0, n-1)$ é empilhado.

Agora suponha que $n \geq 2$ e que a função `partition` do `quicksort` particiona o vetor `a[]` em dois subvetores de tamanhos a e b, onde $a + b = n - 1$ (descarta-se a posição i) e $0 \leq a \leq b$.

Como a parte menor de tamanho a é ordenada primeiro, a pilha conterá antes da ordenação da parte maior no máximo $t(a) + 1$ pares (o "+1" deve-se ao par com os extremos da parte de tamanho b). Na ordenação da parte maior de tamanho b, a pilha conterá no máximo $t(b)$ pares. Segue-se então que:

$$t(n) \leq \max \left\{ \max\{t(a)+1,\, t(b)\} \;\; a + b = n - 1 \text{ e } 0 \leq a \leq b \right\}$$

para $n \geq 2$.

(iii)

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
s_n	1	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5	5

(iv)

Vamos provar por indução que $s_n = \lfloor \log n \rfloor + 1$, para $n \geq 1$ (logaritmo na base 2).

Base: para $n = 1$ temos que $s_1 = 1 = \lfloor \log 1 \rfloor + 1$.

Passo de indução: suponha que $s_n = \lfloor \log n \rfloor + 1$, para $1 \leq n \leq k-1$. Vamos provar que $s_k = \lfloor \log k \rfloor + 1$.

Temos que $s_k = s_{\lfloor k/2 \rfloor} + 1$ por definição. Aplicando a hipótese de indução para $\lfloor k/2 \rfloor$ temos que:

$$s_k = s_{\lfloor k/2 \rfloor} + 1 = \lfloor \log k/2 \rfloor + 1 + 1.$$

Se k for uma potência de 2, $\lfloor \log k/2 \rfloor = \log k/2 = \log k - 1$, e portanto, $s_k = \log k + 1 = \lfloor \log k \rfloor + 1$. Senão, $\lfloor \log k/2 \rfloor = \lfloor \log k \rfloor - 1$ e também segue que $s_k = \log k + 1 = \lfloor \log k \rfloor + 1$.

(v)

Base: temos que $t(0) = s_0$ e $t(1) = s_1$.

Passo de indução: suponha que $t(n) \leq s_n$ para $0 \leq n \leq k - 1$. Vamos provar que $t(k) \leq s_k$.

Pelo item *(ii)*, temos que

$$t(k) \leq \max \left\{ \max\{t(a)+1,\, t(b)\} \;\; a + b = k - 1 \text{ e } 0 \leq a \leq b \right\}.$$

Sejam a, b inteiros que maximizam o lado direito da desigualdade acima. Como $a < k$ e $b < k$, temos por indução que $t(a) \leq s_a = \lfloor \log a \rfloor + 1$ e $t(b) \leq s_b = \lfloor \log b \rfloor + 1$.

Como $a \leq \lfloor k/2 \rfloor$ temos que

$$t(a) + 1 \leq \lfloor \log k/2 \rfloor + 2 = s_k,$$

e como $b < k$ temos que

$$t(b) \leq \lfloor \log k \rfloor + 1 = s_k.$$

Portanto,

$$t(k) \leq \max\{t(a) + 1, t(b)\} \leq s_k.$$

(vi)

Para cada n considere a instância `a[]` com os números $1, \ldots, n$ em ordem crescente. A função `partition` não altera o vetor e se ela devolver sempre a primeira posição (nas eventuais chamadas recursivas), então como o par com os extremos da parte menor é empilhada primeiro, a pilha conterá n pares em algum instante. O espaço necessário para a pilha nestas instâncias é $O(n)$.

Resposta da questão 3.

Descrevemos aqui duas soluções. Naturalmente, na prova bastaria uma delas.

Primeira solução.

A estrutura de dados que utilizaremos é uma *fila de prioridade* Q, onde os elementos desta serão as k listas. A *chave* de cada elemento (lista) é o primeiro inteiro da lista. A fila deve ser capaz de realizar as seguintes operações:

- construir a fila Q em tempo $O(k)$ (ou $O(n \log k)$);
- remover o primeiro inteiro da lista com menor prioridade na fila Q e atualizar a fila Q resultante (caso a lista fique vazia, ela é removida de Q), em tempo $O(\log k)$.

A fila de prioridade Q pode ser implementada como um *heap binário*. A construção de Q pode ser feita usando a função `Build_Heap`, enquanto a segunda operação pode ser implementada usando a função `fixDown` (como no heapsort). As duas operações podem ser executadas com as complexidades especificadas.

Suponha que as listas são fornecidas em um vetor `V`. O algoritmo para juntar as listas ordenadas fica assim:

```
Junta_Ordena(V[], k)
  {
    Q = Constrói_Fila(V,k);
    L = lista_vazia();
    while (Q não vazia) {
      x = Remove(Q);
      insere(L,x);
    }
    return L;
  }
```

A função `Constrói_Fila` constrói a fila Q em tempo $O(k)$ e a lista `L` é inicializada como a lista vazia (ao final, `L` será a lista ordenada desejada).

A função `Remove` implementa a segunda operação que Q deve suportar e consome tempo $O(\log k)$. Observe que ela devolve o menor inteiro em todas as listas que ainda estão em Q. A função `insere(Q,x)` insere o inteiro `x` no fim da lista `L` e consome tempo constante. O laço `while (Q não vazia)` é executado n vezes.

Assim, a complexidade de `Junta_Ordena` é $O(k + n \log k)$. Claramente, ela devolve uma lista ordenada e contém todos os inteiros das k listas originais.

Segunda solução

A segunda idéia consiste em imaginar que as k listas são soluções parciais do mergesort e aplicar a estratégia do mergesort para obter a lista final ordenada. O algoritmo consiste em fazer *merge*s das listas duas a duas até obtermos uma única lista ordenada. Vamos supor que temos a nossa disposição uma função `merge`(L, R) que recebe duas listas de inteiros L e R que faz o *merge* dessas e devolve a lista resultante, em tempo $O(|L| + |R|)$, onde $|L|$ e $|R|$ são os comprimentos das listas L e R.

Suponha que as listas são fornecidas em um vetor `V`. O algoritmo para juntar as listas ordenadas fica assim:

```
Junta_Ordena(V[], k)
  {
    for (m = 1; m < k ; m = 2*m)
      for (i = 0; i < k-m; i = i+2*m)
        V[i] = merge(V[i], V[i+m]);
    return V[0];
  }
```

O `for` externo é executado $O(\log k)$ vezes. Em cada uma dessas iterações o tempo gasto total gasto para fazer todos os *merge*s dentro do `for` interno é $O(n)$. Portanto, a complexidade do algoritmo é $O(n \log k)$. Claramente, a lista devolvida está ordenada e contém todos os inteiros das k listas originais.

Resposta da questão 4.

1ª parte.

Nesta questão o vetor `a[]` contém os números $\{0, \ldots, n-1\}$.

(i)

Para $a[r] = k$ a troca `exch(a[i], a[last])` é executada $n - k - 1$ vezes.

(ii)

Dado que cada permutação é equiprovável, o número esperado de trocas `exch(a[i], a[last])` executadas é

$$\sum_{k=0}^{n-1} (n-k-1)\Pr[a[r]=k] \;=\; \sum_{k=0}^{n-1} \frac{(n-k-1)}{n} \;=\; \frac{n-1}{2}.$$

2ª parte.

(i)

O número total de comparações envolvendo elementos de `a[]` é $n+1$, se $a[r] \neq 0$, ou n, se $a[r] = 0$.

(ii)

Suponha que $\texttt{a}[\texttt{r}] = k$ e seja q o número de índices i $(0 \leq i < k)$ tais que $a[i] > k$. O número de vezes que executamos a troca `exch(a[i], a[j])` é q.

(iii)

Dado que todas as permutações são equiprováveis, e *supondo* que $a[r] = k$, o valor esperado de q no item anterior é

$$
\begin{aligned}
q_k = \sum_{t=0}^{k} t\Pr[q = t] &= \sum_{t=0}^{k} t\binom{n-k-1}{t}\binom{k}{k-t}\binom{n-1}{k}^{-1} \\
&= \binom{n-1}{k}^{-1} \sum_{t=1}^{k} t\binom{n-k-1}{t}\binom{k}{t} \\
&= \binom{n-1}{k}^{-1} \sum_{t=1}^{k} k\binom{n-k-1}{t}\binom{k-1}{t-1} \\
&= k\binom{n-1}{k}^{-1}\binom{n-2}{n-k-2} \\
&= \frac{k(n-1-k)}{n-1}.
\end{aligned}
$$

(iv)

Supondo que todas as permutações são equiprováveis, o número esperado de trocas `exch(a[i], a[j])` executadas é

$$
\begin{aligned}
\sum_{k=0}^{n-1} q_k \Pr[a[r] = k] &= \sum_{k=0}^{n-1} \frac{k(n-1-k)}{n-1}\frac{1}{n} \\
&= \frac{1}{n}\sum_{k=0}^{n-1} k - \frac{1}{n(n-1)}\sum_{k=0}^{n-1} k^2 \\
&= \frac{1}{n}\frac{n(n-1)}{2} - \frac{1}{n(n-1)}\frac{(n-1)n(2n-1)}{6} \\
&= \frac{n-2}{6}.
\end{aligned}
$$