Questão 1

a. Apesar de algumas inconsistências para diferentes execuções, verificou-se que o algoritmo de Dijkstra em média foi o mais eficiente ao se considerar o tempo de viagem entre Camaragibe e Cajueiro Seco.

```
O algoritmo mais rápido foi: Dijkstra com 0.00003457 segundos

O algoritmo mais rápido com o grafo não-ponderado foi: DFS com 0.00002003 segundos
```

b. Para um grafo não-ponderado, o método ideal é a busca em largura (BFS), pois, uma vez que os pesos das arestas são iguais em todo o grafo, esse algoritmo retornará o menor caminho entre dois nós de uma forma mais rápida e direta através da verificação de nós visitados, em comparação com Dijkstra que naturalmente faz operações mais custosas computacionalmente, como as atualizações das distâncias em relação ao peso de cada aresta, e troca dos valores na fila de prioridade. Já a busca em profundidade (DFS), percorre todo o caminho de um nó antes de partir para o vizinho, isso pode fazer com que o caminho mais curto demore mais para ser encontrado.

c. A proposta é de uma nova conexão entre Tancredo Neves e Alto do Céu, o que diminuiria o tempo de viagem de 74 min para 62 min de acordo com o nosso programa.

Questão 2

a. De acordo com o programa implementado.

b. De acordo com o programa implementado.

c. O tempo varia de acordo com cada execução, como pode ser testado ao rodar o programa. Um exemplo de execução retorna os seguintes valores:

```
Tempo de execução do Merge Sort: 0.0003905296 segundos
Tempo de execução do Quick Sort: 0.0018317699 segundos
Tempo de execução do Heap Sort: 0.0012888908 segundos
O melhor método é mergesort com tempo de execução de 0.0003905296 segundos
```

Para execuções sucessivas, o mesmo resultado final é alcançado, a saber: o método MergeSort se mostra o mais eficiente entre os três no contexto do problema.

d. A análise de complexidade revela que:

i. Para o algoritmo *MergeSort*, sua complexidade temporal tem sempre valor **O(n log n)**, independentemente do cenário (melhor, médio ou pior). Já sua complexidade espacial tem valor **O(n)**, devido ao método realizar uma separação recursiva da lista em duas metades e mesclá-las novamente de forma ordenada, o que exige espaço adicional para armazenamento. Essa sua natureza permite que ele seja uma boa escolha para ordenar grandes conjuntos de dados, uma vez que oferece estabilidade, facilidade de implementação e realiza a ordenação paralelamente. No entanto, a necessidade de memória adicional para armazenar listas mescladas pode ser um problema, além do fato de que ele é mais lento em relação ao *QuickSort*.

ii. O algoritmo *QuickSort* possui complexidade temporal e espacial diferentes para os casos melhor/médio e pior, a saber: temporal com valor de **O(n log n)** nos casos melhor/médio e **O(n²)** no pior deles; e espacial com valores **O(log n)** nos casos melhor/médio e **O(n)** no pior caso. Isso ocorre devido à influência exercida pela escolha do *pivot* a ser usado para particionar a lista, o que ocasionar numa recursão desequilibrada. Portanto, o algoritmo é eficiente para grandes conjuntos de dados e requer pouca memória para ser implementado. Por outro lado, não é muito efetivo para conjuntos pequenos e a má escolha do *pivot* gera problemas no tempo de execução, o que pode provocar instabilidades.

iii. O algoritmo *HeapSort* possui complexidade temporal de **O(n log n)**, devido ao tempo de criação e extração do heap, e complexidade espacial **O(1)**, pois não exige a criação de espaço adicional proporcional ao de entrada para sua implementação. Por ter uma complexidade temporal constante, é considerado um bom método para ordenar grandes conjuntos, além de requerer pouca ou nenhuma memória adicional para que funcione. Contudo, não é tão eficiente e possui instabilidades que podem torná-lo indesejável.

Considerando todos esses fatores, o melhor método para o caso da lista de eventos seria o MergeSort, uma vez que ter estabilidade seria adequado

quando se trata de organizar eventos programados, com data e hora específicas, e reordenar quando houver alterações na programação.

e. Um bom sistema de ordenação oferece aos turistas a melhor forma de identificar quais eventos eles desejam ir, para que se programem antecipadamente e evitem atrasos. Além disso, caso ocorram alterações eles não serão pegos de surpresa, devido às atualizações imediatas e consistentes que o sistema pode oferecer.