2.

Para encontrar o maior elemento, basta comparar o elemento A[1] com B[1], o maior será o maior elemento do conjunto. Para encontrar o segundo maior elemento, basta comparar o elemento A[1] com B[1]. Se o A[1] for o maior, basta compara o A[2] com o B[1], senão serão comparados os elementos A[1] com B[2]. Neste caso, serão necessárias 2 comparações. Para encontrar o *n*-ésimo maior elemento, basta realizar *n* comparações.

```
int main() {
     int i, ia, ib, x, n;
     int A[10]={ . . . };
     int B[10]={ . . . };
     A[11] = -∞;
     B[11] = -∞;
     cout<<"\nEntre com o valor de n p/ achar o n-esimo maior elemento:";
     cin>>n;
     if (n <= |A| + |B|) {
        ia = 1;
        ib = 1;
        for (i = 0; i < n; i++)
           if (A[ia] > B[ib]) {
              x = A[ia];
              ia++;
           }
           else {
              x = B[ib];
              ib++;
           }
     }

     cout<<i<<"-esimo maior elemento = "<<x;

}
```

2 3.

a) Existem `n+1` lugares possíveis para inserir este novo elemento.
Melhor caso: elemento será inserido após o elemento que está na posição $\lceil n/2 \rceil$.
Pior caso: elemento será inserido na posição `1` ou `n+1`. Para inserir um novo elemento serão necessárias, no pior caso `log(n + 1)` comparações.

b) `x` é o elemento a ser inserido. O algoritmo é ótimo.

```
esq = 1;
dir = n;
achou = false;
while (!achou && esq < dir){
   i = (esq + dir)/2;
   if (A[i] >= x)
      if (A[i+1] <= x)
         achou = true;
      else
         esq = i+1;
   else
      dir = i;
}
if (achou)
   cout<<"\nSera inserido na posicao "<<i+1;
else {
   i = (esq + dir)/2;
   if (x > A[i])
      cout<<"\nSera inserido na posicao "<<i;
   else
      cout<<"\nSera inserido na posicao "<<i+1;
}
```

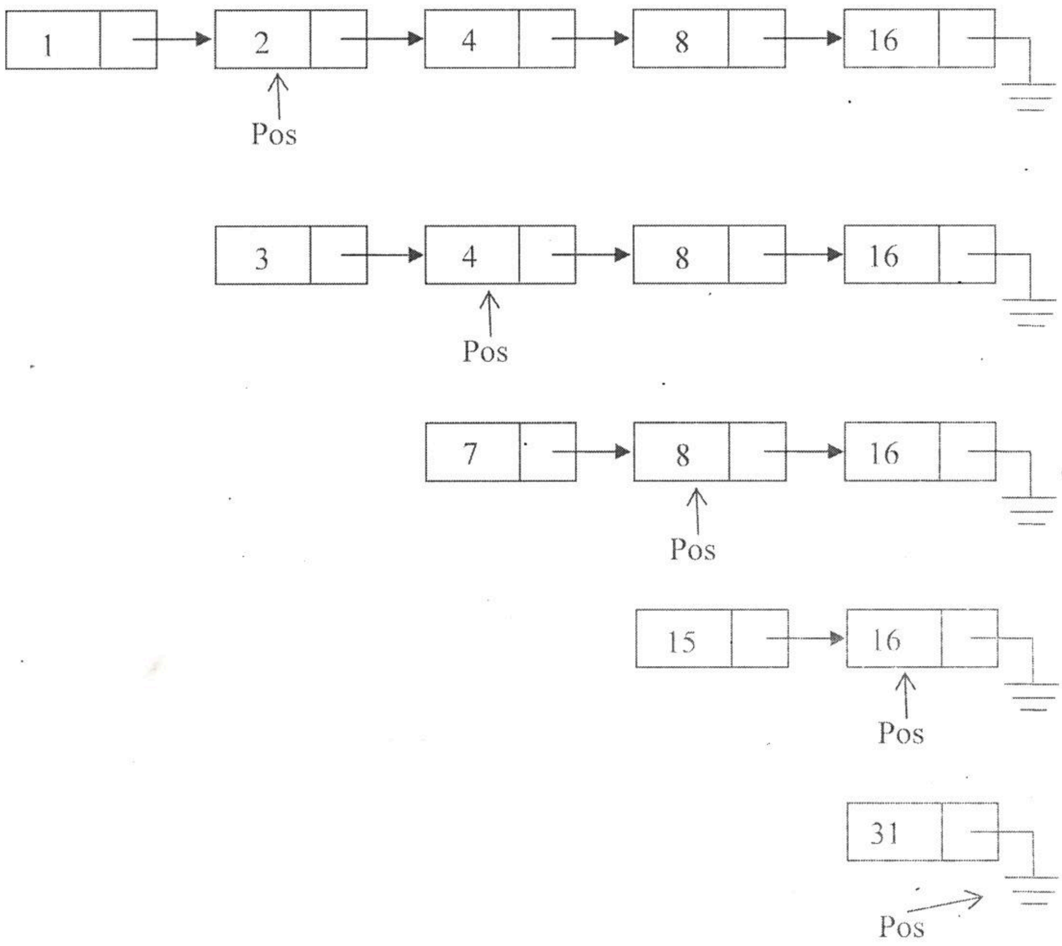
3 4.

a) A implementação desse algoritmo foi feita utilizando a estrutura de dados lista encadeada sem célula cabeça. Abaixo se encontra a parte central do algoritmo. A idéia básica é utilizar um ponteiro para indicar a posição de inserção da soma recém calculada. Como a i-ésima soma será sempre maior ou igual à i-ésima–1, o ponteiro descrito anteriormente nunca precisará ser decrementado. Com essa idéia conseguimos uma implementação em tempo linear.

```
...
Pos = Lista.Primeiro->Prox;
for (i = 0; i < Lista.Tamanho-1; i++) {
   Soma = Lista.Primeiro->Item + Lista.Primeiro->Prox->Item;
   while((Pos->Prox != NULL) && (Pos->Prox->Item < Soma))
      Pos = Pos->Prox;
   Aux = Pos->Prox;
   Pos->Prox = new(TipoCelula);
   Pos = Pos->Prox;
   Pos->Item = Soma;
   Pos->Prox = Aux;
   Aux = Lista.Primeiro;
   Lista.Primeiro = Lista.Primeiro->Prox;
   delete(Aux);
```

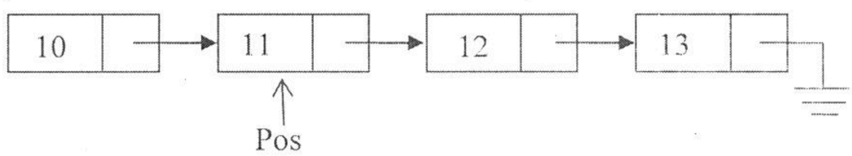
```
    Aux = Lista.Primeiro;
    Lista.Primeiro = Lista.Primeiro->Prox;
    delete(Aux);
}
...
```

b) Considerando que a medida de complexidade relevante seja o número de vezes que devemos incrementar o valor de Pos para encontrar a posição de inserção, o melhor caso acontece quando a cada soma realizada, o ponteiro é incrementado 1 posição (pelo menos 1 incremento por soma é necessário).



$T(n) = n-1$

c) Considerando que a medida de complexidade relevante seja o número de vezes que devemos incrementar o valor de Pos para encontrar a posição de inserção, o pior caso ocorre quando o resultado da primeira soma calculada é maior que o maior número presente na lista. Como exemplo, temos:



Neste exemplo, `Soma = 21` que é maior que 13. Então, na primeira iteração Pos andará n-2 casas. Para as n-2 somas restantes, Pos andará 1 posição para cada uma. Sendo assim, no pior caso $T(n) = \underbrace{(n-2)}_{1^{a}\text{ iteração}} + \underbrace{(n-2)}_{(n-2)\text{ iterações restantes}} = 2n-4 \Rightarrow T(n)=O(n)$

d) Não. Para executarmos a unificação de listas deveremos somar n elementos, para isso são necessárias (n-1) operações. Logo, a unificação de listas só pode ser executada com complexidade no mínimo linear.