EXAMEN FINAL - ALGORITMIA

UNIVERSIDAD TECNOLÓGICA DE PEREIRA
Programa de Maestría en Ingeniería de Sistemas y Computación
Profesor Hugo Humberto Morales Peña
Sábado 24 de Enero de 2015

1. **COLAS DE PRIORIDAD [40 puntos]**

   Heapsort es un algoritmo excelente, pero el Quicksort es mejor en la práctica. Sin embargo, la estructura de datos montón (heap) tiene muchos usos. El uso más importante que se le da a la estructura de datos montón es para implementar de forma eficiente colas de prioridad (Priority Queue). Como en los montones, las colas de prioridad son de dos tipos: colas de máxima prioridad y colas de mínima prioridad. En este examen trabajaremos las colas de mínima prioridad.

   En una cola de mínima prioridad el elemento más pequeño es el que se atiende primero.

   Una cola de mínima prioridad soporta las siguientes operaciones:

   - **MinPQ_Insert**(Q, x): Inserta el elemento x en la cola de mínima prioridad Q.
   - **MinPQ_Minimum**(Q): Retorna el elemento más pequeño de la cola de mínima prioridad Q.
   - **MinPQ_Extract**(Q): Remueve y retorna el elemento más pequeño de la cola de mínima prioridad Q.
   - **MinPQ_DecreaseKey**(Q, i, k): Disminuye el valor del elemento ubicado en la posición i de la cola de prioridad Q al nuevo valor k. Se asume que el valor de k como máximo es el valor del elemento ubicado en la posición i de Q.

   Las funciones que se necesitan del manejo de montones mínimos en colas de mínima prioridad son:

   function Parent(i)
   1. **return** $\lfloor \frac{i}{2} \rfloor$

   function Left(i)
   1. **return** $2 * i$

   function Right(i)
   1. **return** $2 * i + 1$

   function MinHeapify(Q, i)
   1. $l =$ **Left**(i)
   2. $r =$ **Right**(i)
   3. **if** $l \leq heapSize$ **and** $Q[l] < Q[i]$
   4. $\quad least = l$
   5. **else** $least = i$
   6. **if** $r \leq heapSize$ **and** $Q[r] < Q[least]$
   7. $\quad least = r$
   8. **if** $least \neq i$
   9. $\quad$ **exchange** $Q[i]$ **with** $Q[least]$
   10. $\quad$ **MinHeapify**(Q, $least$)

   Las funciones que soportar las operaciones del manejo de colas de mínima prioridad son:

   function MinPQ_Minimum(Q)
   1. **return** $Q[1]$

   function MinPQ_Extract(Q)
   1. **if** $heapSize < 1$
   2. $\quad$ **error** "heap underflow"
   3. $min = Q[1]$
   4. $Q[1] = Q[heapSize]$
   5. $heapSize = heapSize - 1$
   6. **MinHeapify**(Q, 1)
   7. **return** min

function MinPQ_DecreaseKey(Q, i, key)

1. **if** $key > Q[i]$
2. **error** "new key is higher than current key"
3. $Q[i] = key$
4. **while** $i > 1$ **and** $Q[$**Parent**$(i)] > Q[i]$
5. **exchange** $Q[i]$ **with** $Q[$**Parent**$(i)]$
6. $i =$ **Parent**(i)

function MinPQ_Insert(Q, key)

1. $heapSize = heapSize + 1$
2. $Q[heapSize] = \infty$
3. **MinPQ_DecreaseKey**(Q, $heapSize$, key)

La función **MinPQ_Minimum** tiene una complejidad en tiempo de ejecución de $O(1)$. Las funciones **MinPQ_Extract**, **MinPQ_DecreaseKey** y **MinPQ_Insert** tienen complejidad en tiempo de ejecución de $O(log\ n)$.

Realizar el siguiente trabajo asumiendo que $heapSize$ es una variable global la cual es inicializada en cero y que la cola de mínima prioridad es almacenada en el vector Q.

a) **[2.5 puntos]** MinPQ_Insert(Q, 5)
b) **[2.5 puntos]** MinPQ_Insert(Q, 4)
c) **[2.5 puntos]** MinPQ_Insert(Q, 3)
d) **[2.5 puntos]** MinPQ_Insert(Q, 2)
e) **[2.5 puntos]** MinPQ_Insert(Q, 1)
f) **[2.5 puntos]** MinPQ_Extract(Q)
g) **[2.5 puntos]** MinPQ_Extract(Q)
h) **[2.5 puntos]** MinPQ_Insert(Q, 3)
i) **[2.5 puntos]** MinPQ_Extract(Q)
j) **[2.5 puntos]** MinPQ_Extract(Q)
k) **[2.5 puntos]** MinPQ_Insert(Q, 6)
l) **[2.5 puntos]** MinPQ_Extract(Q)
m) **[2.5 puntos]** MinPQ_Extract(Q)
n) **[2.5 puntos]** MinPQ_Insert(Q, 9)
ñ) **[2.5 puntos]** MinPQ_Extract(Q)
o) **[2.5 puntos]** MinPQ_Extract(Q)

2. ANÁLISIS Y DISEÑO [60 puntos]

Sumar Todos Los Números En Un Conjunto

Si!, el nombre del problema refleja el trabajo a realizar; es justamente sumar un conjunto de números. Usted puede sentirse motivado a escribir un programa en su lenguaje de programación favorito para sumar todo el conjunto de números. Tal problema debe ser algo muy sencillo para su nivel de programación. Por este motivo le vamos a poner un poco de sabor y vamos a volver el problema mucho más interesante.

Ahora, la operación de suma tiene un costo, y el costo es el resultado de sumar dos números. Por lo tanto, el costo de sumar 1 y 10 es 11. Por ejemplo, si usted quiere sumar los números 1, 2 y 3, hay tres formas de hacerlo:

- Forma 1:
  1 + 2 = 3, costo = 3
  3 + 3 = 6, costo = 6
  Costo total = 9
- Forma 2:
  1 + 3 = 4, costo = 4
  2 + 4 = 6, costo = 6
  Costo total = 10
- Forma 3:
  2 + 3 = 5, costo = 5
  1 + 5 = 6, costo = 6
  Costo total = 11

Yo creo que usted ya ha entendido su misión, sumar un conjunto de números enteros tal que el costo de la operación suma sea mínimo.

Entrada

Cada caso de prueba debe comenzar con un número entero positivo N ($2 \leq N \leq 5000$), seguido por N números enteros positivos (todos son menores o iguales a 100000). La entrada es finalizada por un caso donde el valor de N es igual a cero. Esta caso no debe ser procesado.

Salida

Para cada caso de prueba imprimir una sola línea con un número entero positivo que representa el costo total mínimo de la operación suma después de sumar los N números.

Ejemplo de Entrada

```
3
1 2 3
5
5 4 3 2 1
0
```

Ejemplo de Salida

```
9
33
```

Trabajo a Realizar - 60 puntos

Construir una programa en pseudo código que resuelva el problema anterior con una complejidad en tiempo de ejecución de $\Theta(n\ log\ n)$ y una complejidad en espacio de almacenamiento de $\Theta(n)$