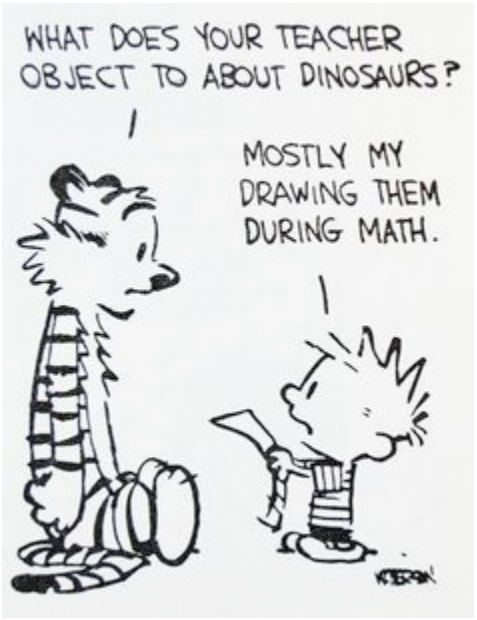
CS 302 – Assignment #8

Purpose: Learn to about priority queues.
Due: Tuesday (10/31)
Points: 125 Part A → 75 pts, Part B → 50 pts

Assignment - Part A:

In recreational mathematics, a perfect power[1] is a positive integer that can be expressed as an integer power of another positive integer. More formally, ***n*** is a perfect power if there exists natural numbers ***m*** > 1, and ***k*** > 1 such that $m^k = n$. In this case, ***n*** may be called a perfect ***k***th power. For example, 4 and 81 are perfect powers since $4 = 2^2$ and $81 = 3^4$. In order to ensure overall efficiency, we will use a fast integer exponentiation algorithm[2].

One algorithm for finding perfect powers uses a priority queue. As such, you should develop a C++ class to implement a priority queue.



Once the priority queue is developed and tested, create a main program to use the priority queue to find and list the perfect powers (in numeric order). Write a main program that reads in a command line parameter, ***limit***, and prints perfect powers such that ***m***, and ***k*** are > 1 and ***n*** is less than or equal to ***limit***. The ***limit*** should be read from the command line and must be verified to be between 10 and 1,000,000 (inclusive). All values should be of type **`unsigned long long`**. If the limit specifier or limit value are invalid, an appropriate error message should be displayed and the program terminated. Refer to the example executions for formatting.

Note, this priority queue may be used on a future assignment.

Part B:

When completed, create and submit a write-up (PDF format) not too exceed ~750 words including the following:

- Name, Assignment, Section
- Summary of the implemented priority queue data structure.
- Big-Oh for the priority queue operations (insert, deleteMin, reheapUp, reheapDown, buildHeap, and printHeap). Include an explanation of the buildHeap result.
- Big-Oh for the integer exponentiation function.
- Explain the difference between repeated inserting and using buildHeap.

Submission:

- Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 23:50.

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

---

1 For more information, refer to: https://en.wikipedia.org/wiki/Perfect_power
2 For more information, refer to: https://en.wikipedia.org/wiki/Exponentiation_by_squaring

Perfect Powers Algorithm

Algorithm to find solutions of $n = a^b$ such that ***a*** and ***b*** are > 1 and ***n*** is ≤ ***limit***. Assuming the use of a structure, ***numberPair***, containing ***a*** and ***b*** and that priority is ***n*** (where ***n*** = a^b). All values should be of type **`unsigned long long`**.

insert (2,2) with priority of 4 into queue.
loop while top of priority queue is ≤ ***limit***.
    remove operation → (***a***, ***b***) ***priority***
    if priority not the same as previous priority value
        output priority
    insert (***a***+1, ***b***) with priority of (***a***+1)^***b***
    if (***a*** == 2)
        insert (***a***, ***b***+1) with priority of ***a***^(***b***+1)

In order for this to be effective, a fast recursive integer exponentiation function should be used (as outlined on the Wikipedia page, https://en.wikipedia.org/wiki/Exponentiation_by_squaring). The algorithm is as follows:

```
function exp_by_squaring(x, n)
      if n = 0  then return 1;
      else if n = 1  then return x ;
      else if n is even  then return exp_by_squaring(x*x,  n/2);
      else if n is odd  then return x * exp_by_squaring(x*x, (n-1)/2);
```

The function **must** be recursive. Refer to the Wikipedia page for additional explanation. *Note*, you will be asked why this is a faster exponentiation algorithm (as compared to the typical looped approach).

Class Descriptions

- Priority Queue Class
  The priority queue template class will implement functions specified below.

priorityQueue<myType>
-heapNode: struct
-priority: unsigned long long
-item: myType
-count: int
-heapSize: int
-*myHeap: heapNode
+priorityQueue(int=5000)
+~priorityQueue()
+entries() const: int
+insert(const myType, const unsigned long long): void
+peek() const: unsigned long long
+deleteMin(myType &, unsigned long long &): bool
+isEmpty() const: bool
+printHeap() const: void
+readData(const string): bool

-reheapUp(int): void
-reheapDown(int): void
-buildHeap(): void
-resize(): void

Function Descriptions

- The *priorityQueue()* constructor should initialize the binary heap to an empty state. The parameter must be checked to ensure it is at least 5000 and, if not, use the default value.
- The *~priorityQueue()* destructor should delete the heap.
- The *entries()* function should return the total count of elements in the heap.
- The *insert()* function should insert an entry into the binary heap. If the count of heap entries exceeds the heap size, the heap must be expanded via the private *resize()* function. The heap properties must be maintained via the private *reheapUp()* function. The count should be updated.
- The private *buildHeap()* function should update the heap to apply the heap properties.
- The *isEmpty()* function should return true if there are no elements in the heap and false otherwise.
- The *deleteMin()* function should remove the minimum entry from the heap. The heap properties must be maintained via the private *reheapDown()* function. Additionally, the count should be updated. If the heap is already empty, the function should return false and otherwise return the minimum priority information (via reference) and return true.
- The *peek()* function should return a copy of the current minimum entry form the heap. The heap should not be changed in any way.
- The *printHeap()* function should print the current heap in level order with a blank line between each level. Refer to the sample output for an example of the formatting.
- The *readData()* function should read a formatted file (name and priority) and enter the values read into the array **without** using the *insert()* function. The size will need to be checked and the *resize()* function called as needed. Once the values are read, the *buildHeap()* function should be called. The trade-offs associated with this process should be noted in the write-up.
- The *reheapUp()* function to recursively ensure the heap order property is maintained. Starts at tree leaf and works up to the root. Must be written recursively.
- The *reheapDown()* function to recursively ensure the heap order property is maintained. Starts at the passed node and works down to the applicable leaf. Must be written recursively.
- The *resize()* function should create a new heap array twice the size of the existing heap, copy all entries from the current heap into the new heap, and delete the old heap. The *heapSize* should be updated accordingly.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 302 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

Test Script

A test script which will be used for scoring the final submission is provided for reference.

Example Execution – Testing Program:

Below is an example program execution for the testing program.

```
ed-vm% ./pqTest
****************************************************************
CS 302 - Assignment #8
Priority Queue Tester


================================================================
Test Set #0
----------------------------------------
PQ Heap (level order):
google  1

amazon  2
newegg  3

apple  4
dell  5
oracle  6
cisco  7

jupiter  8
belkin  9
ebay  10

----------------------------------------
PQ Heap Size: 10
Priority Order:
google  1
amazon  2
newegg  3
apple  4
dell  5
oracle  6
cisco  7
jupiter  8
belkin  9
ebay  10

================================================================
Test Set #1
----------------------------------------
PQ Heap (level order):
g  1

x  1
y  1

o  2
n  5
p  7
d  9

a  3
c  4
j  6
b  8
e  11
f  12
h  13
k  14

m  15
```

```
--------------------------------------
PQ Heap Size: 16
Priority Order:
g  1
x  1
y  1
o  2
a  3
c  4
n  5
j  6
p  7
b  8
d  9
e  11
f  12
h  13
k  14
m  15

======================================================================
Test Set #3
Large test successful.

======================================================================
Test Set #4
Email (1):  leo@MorbimetusVivamus.com
Email (2):  non.cursus@sapien.edu
Email (3):  ut.ipsum.ac@et.org
Email (4):  nec.ante@eteuismodet.edu
Email (5):  netus@nonummy.com
Email (6):  sit.amet.risus@tacitisociosquad.net
Email (7):  nascetur.ridiculus.mus@DonecestNunc.com
Email (8):  sed.dolor.Fusce@ornaretortorat.org
Email (9):  id.risus@non.co.uk
Email (10):  urna.justo@lectus.ca
Email (11):  elementum.at@ipsumPhasellusvitae.org
Email (12):  augue.porttitor.interdum@orciconsectetuer.org
Email (13):  dolor.Fusce.feugiat@leo.ca
Email (14):  purus@amalesuada.org
Email (15):  ac@fermentummetusAenean.ca
Email (16):  dolor.sit@pede.co.uk
Email (17):  tellus.justo@sitamet.org
Email (18):  consectetuer@lorem.edu
Email (19):  magna.a.tortor@aliquetnecimperdiet.ca
Email (20):  Cras.interdum.Nunc@aduiCras.org
Email (21):  urna@Etiam.net
Email (22):  montes@sem.ca
Email (23):  lacinia.mattis.Integer@InfaucibusMorbi.ca
Email (24):  enim.gravida.sit@mollisPhaselluslibero.com
Email (25):  Integer@tinciduntadipiscingMauris.edu
Email (26):  quis@ullamcorperDuis.ca
Email (27):  lacinia.vitae.sodales@Suspendisse.org
Email (28):  semper.tellus@adipiscing.edu
Email (29):  parturient.montes.nascetur@utmi.org
Email (30):  Nullam@mi.com
Email (31):  bibendum.Donec@amet.net
Email (32):  feugiat@faucibusid.com
Email (33):  pellentesque.tellus@Ut.co.uk
Email (34):  ante.Nunc@nuncacmattis.net
Email (35):  erat.vitae.risus@lacusQuisqueimperdiet.ca
Email (36):  et.malesuada.fames@consequatenim.org
Email (37):  eu.odio@vulputateposuerevulputate.com
Email (38):  id.libero.Donec@tinciduntnuncac.ca
Email (39):  a.malesuada@erat.edu
Email (40):  Fusce@Integervulputaterisus.co.uk
Email (41):  molestie.Sed@luctusvulputate.edu
```

```
Email (42):  Maecenas.mi@vestibulumnequesed.org
Email (43):  Sed.eu@PraesentluctusCurabitur.org
Email (44):  ligula.Nullam.enim@consectetuerrhoncus.com
Email (45):  tincidunt.orci@laoreet.co.uk
Email (46):  dictum.placerat.augue@risusQuisque.co.uk
Email (47):  non.justo.Proin@augueeutempor.com
Email (48):  cursus@Proinnisl.edu
Email (49):  tincidunt.nibh.Phasellus@ac.org
Email (50):  a.felis.ullamcorper@estvitaesodales.edu

****************************************************************
Game Over, thank you for playing.
ed-vm%
```

Example Execution:

Below is an example program execution for the main program.
Note 1, one space between each number, no new line until the end.
Note 2, the **`ed-vm%`** is prompt on my machine. Your prompt will be different.

```
ed-vm%./perfectNums
Usage: ./perfectNums -l <limit>
ed-vm%
ed-vm% ./perfectNums -l 4
Error, invalid limit value.
ed-vm%
ed-vm%  ./perfectNums -l 2000000
Error, invalid limit value.
ed-vm%
ed-vm% ./perfectNums -x 200
Error, invalid limit specifier.
ed-vm%

ed-vm%
ed-vm% ./perfect -l 150
CS 302 - Assignment #8
Perfect Powers Program.

4 8 9 16 25 27 32 36 49 64 81 100 121 125 128 144
ed-vm%

ed-vm%
ed-vm% ./perfect -l 10000
CS 302 - Assignment #8
Perfect Powers Program.

4 8 9 16 25 27 32 36 49 64 81 100 121 125 128 144 169 196 216 225 243 256 289
324 343 361 400 441 484 512 529 576 625 676 729 784 841 900 961 1000 1024 1089
1156 1225 1296 1331 1369 1444 1521 1600 1681 1728 1764 1849 1936 2025 2048 2116
2187 2197 2209 2304 2401 2500 2601 2704 2744 2809 2916 3025 3125 3136 3249 3364
3375 3481 3600 3721 3844 3969 4096 4225 4356 4489 4624 4761 4900 4913 5041 5184
5329 5476 5625 5776 5832 5929 6084 6241 6400 6561 6724 6859 6889 7056 7225 7396
7569 7744 7776 7921 8000 8100 8192 8281 8464 8649 8836 9025 9216 9261 9409 9604
9801 10000
ed-vm%
```

For testing, it might be easiest to re-direct the output to a file.