(c) Copyright A. Gerbessiotis. All rights reserved.

Posting this document on the Web, other than its original location at NJIT, is strictly prohibited unless there is an explicit written authorization by its author (name on top line of this first page of this document).

1 Mini-Project Logistics

Warning: Besides the algorithmic-related programming, for some or all of the options you need to provide command-line processing or file-based input/output. If you are not familiar with those topics and requirements we urge you to become so as soon as possible. Thus start familiarizing with them early in the semester. The more you procrastinate the more problems you will potentially face when you integrate these components with the algorithmic-related material.

STEP-1. Read carefully Handout 2 and follow all the requirements specified there; moreover, observe naming conventions, comments and testing as specified in sections 2-4 of Handout 2.

STEP-2. When the archive per Handout 2 guidelines is ready, upload it to moodle and do not forget to press the submit button; the submission timestamp depends on this (submission button).

BEFORE NOON-time of Tue 23rd April, 2019

If you want to get 0 pts ignore Handout 2.

You may do one or two options (or none at all). You may utilize the same language or not. **We provide descriptions that are to the extent possible language independent: thus a reference in Java is a pointer in C or C++ for example.**

OPTION 1. Do the programming part related to Huffman coding in Java, C, or C++.

OPTION 2. Do the programming part related to Kleinberg's HITS, and Google's PageRank algorithms in Java, C, or C++.

2 OPTION 1: Huffman Coding (134 points)

Implement Huffman coding to compress and decompress arbitrary files (such as a .pdf or .jpg or .mp4 or other) as explained in class and in the Subject notes. You must implement your own Binary Heap (aka array with the required Heap property) and its associated operations. Reminder: No HashMaps. Non-compliance to these will automatically gain you a zero. (See Handout 2.) WE SHALL REVIEW YOUR CODE TO VERIFY ADHERENCE TO THESE GUIDELINES.

Suggestion: view byte values as 'characters' in a generic way independent of file type. Do not expect much compression from files that are small or already compressed (eg .jpg). In general for files of size about 5MiB, an implementation should take no more than a few seconds, may be 15 seconds forgivingly. Moreover, you will be writing bit aligned codes into files: practice with bit operations and packing or unpacking bits into/from bytes. If instead of writing bit 1 you write character 1 (in ASCII or UNICODE) this defeats the compression objective. Avoid getting distracted with the Wikipedia code. *Moreover your code should print on screen (standard output) at the end of the compression two numbers: one is the number of bytes of the compressed file, the other number is the total number of bits utilized for the encoding of the original file. That latter number is the sum of products; each product is the frequency of a bytevalue (aka character) and the number of Huffman bits used for its encoding.*

As stated under RULE 1, resolve issues with command-line processing or file-based input/output early in the semester. Never prompt for input as testing would be automated to some degree. So command line processing to the specifications is a MUST. My suggestion is to always read files in binary. If you don't know what i mean, pick a pdf file build a histogram of its bytes add up its frequencies and make sure that sum is equal to the size of the file as reported by an `ls -l` in a UNIX/Linux/OSX system. Write code to copy the file into another file and compare them using md5sum (a Unix/Linux/etc command).

```
 // Encode: henc  filename     -> filename.huf  ; filename     gets erased
 //              x.pdf         -> x.pdf.huf     ; x.pdf        gets erased
 // Decode: hdec  filename.huf -> filename      ; filename.huf gets erased
 //              x.pdf.huf     -> x.pdf         ;  x.pdf.huf   gets erased
 //              if  filename already exists, it gets overwritten
 // Per Handout 2  you should use   hencWXYZ or hdec_WXYZ for henc and hdec
  //   JAVA EXAMPLE testing steps
% cp p1610s19.pdf  file1        // copy into file1
% java henc file1               // Huffman encode ; stdout suppressed (not shown)
% rm -rf file1                  // If not already removed, remove it explicitly
% java hdec file1.huf           // Huffman decode file1.huf
% diff   p1610s19.pdf file1     // If different something went wrong! diff might not work
% md5sum p1610s19.pdf file1     // Should have same signatures; this is better
  //   EXAMPLE C or C++
% cp p1610s19.pdf  file1
% ./henc file1                  // Huffman encode ; stdout suppressed (not shown)
% rm -rf file1
% ./hdec file1.huf
% diff   p1610s19.pdf file1
% md5sum p1610s19.pdf file1
```

Deliverables. Include all implemented functions (source code only) into an archive per Handout 2 guidelines. Command-line execution: do not prompt to read a file-name.

3 OPTION 2: HITS and PageRank implementations (134 points)

Implement Kleinberg's HITS Algorithm, and Google's PageRank algorithm in Java, C, or C++ as explained.

(A) Implement the HITS algorithm as explained in class/Subject notes adhering to the guidelines of Handout 2. Pay attention to the sequence of update operations and the scaling. For an example of the Subject notes, you have output for various initialization vectors. You need to implement class or function `hits` (e.g. `hitsWXYZ`. For an explanation of the arguments see the discussion on PageRank to follow.

```
% java hits iterations initialvalue filename
% ./hits    iterations initialvalue filename
```

(B) Implement Google's PageRank algorithm as explained in class/Subject notes adhering also to the guidelines of Handout 2 and this description. The input for this (and the previous) problem would be a file containing a graph represented through an adjacency list representation. The command-line interface is as follows. First we have the class/binary file (eg pgrk). Next we have an argument that denotes the number of `iterations` if it is a positive integer or an `errorrate` for a negative or zero integer value. The next argument `initialvalue` indicates the common initial values of the vector(s) used. The final argument is a string indicating the `filename` that stores the input graph.

```
% ./pgrk     iterations initialvalue filename  // in fact pgrkWXYZ
% java pgrk  iterations initialvalue filename  // in fact pgrkWXYZ
```

The two algorithms are iterative. In particular, at iteration t all pagerank values are computed using results from iteration $t-1$. The `initialvalue` helps us to set-up the initial values of iteration 0 as needed. Moreover, in PageRank, parameter d would be set to 0.85. The PageRank $PR(A)$ of vertex A depends on the PageRanks of vertices $T_1,\ldots,T_m$ incident to A, i.e. pointing to A; check Subject 7 section 3.6 for more details. The pageranks at iteration t use the pageranks of iteration $t-1$ (synchronous update). Thus PR(A) on the left in the PageRank equation is for iteration t, but all PR (T_i) values are from the previous iteration $t-1$. Be careful and synchronize! In order to run the 'algorithm' we either run it for a fixed number of iterations and `iterations` determines that, or for a fixed `errorrate` (an alias for `iterations`); an `iterations` equal to 0 corresponds to a default errorrate of 10^{-5}. A `-1, -2, etc , -6` for `iterations` becomes an errorrate of $10^{-1},10^{-2},\ldots,10^{-6}$ respectively. At iteration t when all authority/hub/PageRank values have been computed (and auth/hub values scaled) we compare for every vertex the current and the previous iteration values. If the difference is less than **errorrate** for EVERY VERTEX, then and only then can we stop at iteration t.

Argument `initialvalue` sets the initial vector values. If it is `0` they are initialized to 0, if it is `1` they are initialized to 1. If it is `-1` they are initialized to $1/N$, where N is the number of web-pages (vertices of the graph). If it is `-2` they are initialized to $1/\sqrt{N}$. `filename` first.)

Argument `filename` describes the input (directed) graph and it has the following form. The first line contains two numbers: **the number of vertices followed by the number of edges** which is also the number of remaining lines. PAY ATTENTION THAT NUMBER of VERTICES comes first. The sample graph is treacherous: n and m are the same! All vertices are labeled $0,\ldots,N-1$. Expect N to be less than 1,000,000. In each line an edge (i,j) is represented by `i j`. Thus our graph has (directed) edges $(0,2),(0,3),(1,0),(2,1)$. Vector values are printed to 7 decimal digits. If the graph has N GREATER than 10, then the values for `iterations, initialvalue` are automatically set to `0` and `-1` respectively. In such a case the hub/authority/pageranks at the stopping iteration (i.e t) are ONLY shown, one per line. The graph below will be referred to as `samplegraph.txt`

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

The following invocations relate to `samplegraph.txt`, with a fixed number of iterations and the fixed error rate that determines how many iterations will run. Your code should compute for this graph the same rank values (intermediate and final). A sample of the output for the case of $N > 10$ is shown (output truncated to first 4 lines of it).

```
% ./pgrk  15 -1 samplegraph.txt
Base   :  0 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.2500000 P[ 3]=0.2500000
Iter   :  1 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  2 :P[ 0]=0.2500000 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  3 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  4 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  5 :P[ 0]=0.1732344 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  6 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  7 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  8 :P[ 0]=0.1496625 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  9 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   : 10 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 11 :P[ 0]=0.1424245 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 12 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 13 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0970858 P[ 3]=0.0970858
Iter   : 14 :P[ 0]=0.1402020 P[ 1]=0.1200230 P[ 2]=0.0970858 P[ 3]=0.0970858
Iter   : 15 :P[ 0]=0.1395195 P[ 1]=0.1200230 P[ 2]=0.0970858 P[ 3]=0.0970858


% ./pgrk -3 -1 samplegraph.txt
Base   :  0 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.2500000 P[ 3]=0.2500000
Iter   :  1 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  2 :P[ 0]=0.2500000 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  3 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  4 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  5 :P[ 0]=0.1732344 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  6 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  7 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  8 :P[ 0]=0.1496625 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  9 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   : 10 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 11 :P[ 0]=0.1424245 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 12 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 13 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0970858 P[ 3]=0.0970858


% ./pgrk  0  -1 verylargegraph.txt
Iter   : 4
P[ 0]=0.0136364
P[ 1]=0.0194318
P[ 2]=0.0310227
   ... other vertices omitted
```

For the HITS algorithm, you need to print two values not one. Follow the convention of the Subject notes

```
Base   :  0 :A/H[ 0]=0.3333333/0.3333333 A/H[ 1]=0.3333333/0.3333333 A/H[ 2]=0.3333333/0.3333333
Iter   :  1 :A/H[ 0]=0.0000000/0.8320503 A/H[ 1]=0.4472136/0.5547002 A/H[ 2]=0.8944272/0.0000000
```

or for large graphs

```
Iter   : 37
A/H[ 0]=0.0000000/0.0000002
A/H[ 1]=0.0000001/0.0000238
A/H[ 2]=0.0000002/1.0000000
A/H[ 3]=0.0000159/0.0000000
 ...
```

Deliverables. Include source code of all implemented functions or classes in an archive per Handout 2 guidelines. Document bugs; no bug report no partial points.