Algorithms Exam

Sam Auciello

25 February 2013

Question 1:

Pick any two of the following three problems to analyze:

(i) sorting a list of numbers

(ii) the partition problem: given a multiset (a set-like construction that allows multiple copies of the same thing) of integers, is there a way to partition it into two parts such that the sum of integers in each part is the same?

(iii) The convex hull problem: given a set of points (x[i], y[i]), find that list of points that's on the extreme outer "edge", which form a convex polygon containing all the other points.

For each of the two problems that you choose, pick an algorithm to examine and answer the following questions. (In order of decreasing impressiveness, you may choose a good one that you already know and understand, invent one, or search the literate. In any case, be explicit about where your algorithm came from, and explain it clearly.)

(a) Describe what you expect the O() run time to be for the algorithms, and explain why and and on explictly what type of problems (e.g. worse case, average case, etc).

(b) Implement and test the algorithms in a language of your choice. (Do *not* use built-in or library routines for e.g. sorting.)

```
(c) Run your code on various size data sets that you generate,
recording the number of steps (however you choose to define that) the
algorithms take to run. Show explicitly with a plot of the data from
this "numerical experiment" that the O() behavior is as expected.
```

I chose to look at problems (i) and (ii).

Sorting

I've implemented from memory the following three $O(n \log n)$ sorting algorithms in C:

- Quicksort (in place)
- Mergesort
- Heapsort

Quicksort

Quicksort starts by arbitrarily choosing an element of the list and calling it the pivot. It then swaps the other elements of the list around the pivot until all elements less than the pivot are before the pivot and all elements greater than the pivot are after the pivot. It then recursively calls quicksort on the sublists before the pivot and after the pivot. The base case is lists of length less than 2 which are necessarily already sorted. My implementation operates in place and just for fun uses the XOR swap trick that is probably optimized away by the compiler anyway to make it theoretically extra in-place. Since each number must be visited approximately once in each level of recursion (in different branches) and there are $\log n$ levels of recursion because the size is approximately divided in two at each level, the algorithm should run in $O(n \log n)$ time.

Mergesort

Mergesort starts by dividing the list into two equally sized sublists and recursively calling mergesort on each of them. It then merges the two sorted lists together by taking the lower of the first elements of each list until both lists are empty. As with quicksort the base case is lists of length less than 2. As with quicksort, each number must be visited once in each level of recursion and there are $\log n$ levels of recursion so the algorithm should also run in $O(n \log n)$ time.

Heapsort

unlike mergesort and quicksort, heapsort is not recursive. Heapsort makes use of a datastructure called a heap which is a binary tree in which each node is greater than or equal to it's parent. Adding an element to a heap is a $\log n$ operation because the new element must be added at the bottom of the tree and might need to be compared to (and possibly swapped with) each of its ancestors all the way up the tree to find a valid placement. Removing the smallest element of a heap is similarly a $\log n$ operation because the root element must be swapped to the bottom and then the new root needs to be compared to (and possibly swapped with) it's decendants until if finds a valid placement. Heapsort simply takes the initial list and inserts each element into a heap then produces the sorted list by taking the smallest number from the heap until it is empty. Since we need two $\log n$ operations for each element of the list the algorithm should run in $O(n \log n)$ time. There is an efficient way to store a heap in as an array; the values from the tree are simply read out top to bottom left to right into the array. The parent of a node in the array is at the index $\lfloor \frac{i-1}{2} \rfloor$ where i is the index of the node and the children of a node are at $2i + 1$ and $2i + 2$ respectively. According to wikipedia, it is also possible to run heapsort in place by storing the in this manner in the same place as the initial list. In my implementation I used a separate scratch array to store the heap.

Numerical experiment

My code can be tested with:

```
$ cd sorting
$ gcc -Wall test.c -o test && ./test
```

I've sumarized the results below:

```
size: 10
n*log_2(n): 30
quicksort: 23
mergesort: 34
heapsort: 15

size: 20
n*log_2(n): 80
quicksort: 59
```

```
mergesort: 88
heapsort: 61

size: 30
n*log_2(n): 120
quicksort: 143
mergesort: 148
heapsort: 110

size: 40
n*log_2(n): 200
quicksort: 245
mergesort: 216
heapsort: 161

size: 50
n*log_2(n): 250
quicksort: 283
mergesort: 286
heapsort: 212
```

These numbers should be regarded as approximate due to my somwhat arbitrary choice of where to increment the counter. All three algorithms clearly grow with proportion to $n \log n$.

The Partition Problem

I've implemented two versions of a brute force solution to this problem, one in C and and one in Ruby.

In C

My solution in C simply enumerates the powerset of the input multiset until it either reaches the end or finds a solution. It does this in a particularly concise manner by taking advantage of the isomorphism between the powerset of a multiset of size n and the binary expansions of the set of integers that satisfy $0 \leq i < 2^n$. My program simply enumerates the integers from 0 to 2^n, checks to see if the partitioning resulting from that integer has equal sums and returns that integer if it does.

My algorithm should run in $O(2^n)$ time since it is literally enumerating

the powerset. It has the additinal disadvantage of being limited to input multisets no bigger than the log of the largest integer that can be trivially stored in my architecture. In this case 63 since my implementation requires the integer to be signed to allow negative numbers to signify no partitioning having been found. I've run my code on the prepared lists of orders 5, 10, 20, and 50. I specifically used a mix of lists that can and cannot be partitioned to give a better sense of the algorithms behavior. As one might expect, the algorithm runs significantly faster when it finds a solution:

```
$ cd partitions
$ gcc -Wall brute.c -o brute && ./brute
[ 53, 24, 20, 81, 90 ]
[ 53, 81 ] [ 24, 20, 90 ]
2^n:32
steps:10

[ 59, 54, 88, 86, 64 ]
no partitioning exists
2^n:32
steps:32

[ 53, 55, 74, 44, 66, 1, 92, 34, 42, 57 ]
[ 53, 55, 74, 1, 34, 42 ] [ 44, 66, 92, 57 ]
2^n:1024
steps:424

[ 2, 81, 10, 66, 50, 43, 22, 80, 89, 67 ]
no partitioning exists
2^n:1024
steps:1024

[ 0, 0, 13, 0, 36, 40, 23, 38, 38, 25, 53, 37, 85, 10, 51, 1, 74, 71, 33,
  34 ]
[ 13, 36, 40, 38, 38, 25, 53, 37, 51 ] [ 0, 0, 0, 23, 85, 10, 1, 74, 71,
  33, 34 ]
2^n:1048576
steps:20405

[ 87, 68, 97, 55, 26, 48, 62, 48, 43, 80, 75, 13, 25, 62, 76, 39, 3, 80,
  37, 7 ]
```

```
no partitioning exists
2^n:1048576
steps:1048576

[ 67, 78, 63, 45, 55, 13, 7, 75, 37, 84, 68, 68, 54, 4, 9, 90, 70, 58, 39,
  1, 19, 13, 33, 59, 88, 14, 98, 83, 69, 17, 87, 61, 81, 21, 64, 19, 8,
  71, 6, 34, 88, 8, 17, 69, 57, 94, 47, 57, 4, 65 ]
[ 67, 78, 63, 45, 55, 13, 7, 75, 37, 84, 68, 68, 54, 9, 90, 70, 58, 39,
  19, 33, 88, 98 ] [ 4, 1, 13, 59, 14, 83, 69, 17, 87, 61, 81, 21, 64, 19,
  8, 71, 6, 34, 88, 8, 17, 69, 57, 94, 47, 57, 4, 65 ]
2^n:1125899906842624
steps:89645056
```

In ruby

My solution in ruby uses a less technically efficient recursive backtracking search and simply stores the partitioned state using two smaller lists. It makes up for these inefficiencies with two improvements. First it sorts the intial list so that larger numbers will be tried first and then it checks the sums of the two smaller lists at each step. If the difference between the two smaller sums is greater than the sum of the remaining unassigned numbers then there is no solution below this point and the search can backtrack. I used a ruby monkey patch to allow arrays of numbers to keep track of their sums internally as numbers are moved between them to avoid the overhead of recalculating the sum each time. Because it is a backtracking search that goes to a maximum depth of n with a branching ratio of 2 it should run in $O(2^n)$ time. Numerical exeriment indicates that my improvement is cutting this down significantly. My recursive search function is being called roughly between $\frac{2^n}{10}$ and $\frac{2^n}{100}$ times even when no solution is found:

```
$ ruby brute.rb
size: 5
2^n: 32
[ 561, 778, 931, 344, 719 ] can not be partitioned
steps: 13
[ 151, 278, 115, 396, 512 ] can not be partitioned
steps: 9
[ 122, 962, 821, 662, 414 ] can not be partitioned
steps: 9
[ 863, 941, 245, 256, 296 ] can not be partitioned
```

```
steps: 13
[ 670, 315, 177, 915, 997 ] can not be partitioned
steps: 5
 ---
size: 10
2^n: 1024
[ 284, 544, 463, 649, 761, 120, 501, 484, 92, 41 ] can not be partitioned
steps: 113
[ 461, 695, 579, 443, 284, 624, 871, 319, 517, 874 ] can not be partitioned
steps: 263
[ 306, 321, 873, 572, 999, 358, 569, 619, 597, 957 ] can not be partitioned
steps: 255
[ 514, 758, 276, 603, 938, 219, 840, 622, 36, 120 ] can not be partitioned
steps: 105
[ 469, 871, 605, 731, 691, 68, 610, 966, 544, 193 ] can be partitioned as
[ 966, 691, 605, 544, 68 ] [ 871, 731, 610, 469, 193 ]
steps: 131
 ---
size: 20
2^n: 1048576
[ 61, 764, 989, 36, 791, 271, 456, 274, 61, 672, 948, 82, 677, 605, 554,
  314, 816, 949, 714, 225 ] can not be partitioned
steps: 37827
[ 644, 129, 214, 119, 378, 552, 705, 539, 611, 943, 222, 340, 173, 32,
  312, 911, 689, 526, 330, 568 ] can not be partitioned
steps: 61741
[ 834, 16, 154, 318, 901, 645, 506, 751, 496, 865, 733, 935, 779, 293,
  376, 949, 220, 626, 269, 641 ] can not be partitioned
steps: 64287
[ 618, 725, 998, 999, 520, 260, 409, 663, 731, 424, 889, 580, 582, 620,
  548, 659, 953, 422, 712, 951 ] can not be partitioned
steps: 179453
[ 485, 710, 299, 615, 412, 151, 630, 333, 373, 195, 78, 887, 920, 666,
  783, 256, 318, 465, 791, 766 ] can not be partitioned
steps: 87827
 ---
size: 25
2^n: 33554432
[ 68, 767, 397, 164, 873, 994, 135, 246, 708, 329, 172, 612, 352, 370,
  820, 841, 31, 36, 722, 477, 961, 18, 85, 66, 285 ] can not be partitioned
```

```
steps: 618285
[ 482, 566, 270, 370, 714, 643, 337, 340, 939, 112, 943, 367, 199, 789,
  320, 19, 727, 390, 440, 607, 323, 216, 878, 987, 721 ] can not be partitioned
steps: 1715443
[ 48, 552, 458, 16, 207, 749, 248, 359, 265, 505, 393, 731, 941, 981, 562,
266, 476, 555, 671, 501, 504, 909, 188, 953, 929 ] can not be partitioned
steps: 1300743
[ 395, 165, 443, 767, 513, 590, 613, 518, 514, 70, 328, 887, 679, 135,
  979, 939, 149, 83, 990, 669, 545, 521, 409, 772, 924 ] can not be partitioned
steps: 1687875
[ 953, 176, 401, 934, 954, 961, 744, 133, 154, 729, 539, 536, 951, 136,
  489, 493, 925, 208, 699, 955, 555, 255, 671, 736, 399 ] can be partitioned as
[ 961, 955, 954, 953, 951, 934, 925, 401, 176, 133 ] [ 744, 736, 729, 699,
  671, 555, 539, 536, 493, 489, 399, 255, 208, 154, 136 ]
steps: 260
 ---
```

Question 2:

```
In a language of your choice, illustrate a depth-first and
breadth-first tree search, preferably using a stack and a queue,
for a small "sliding block" puzzle.

A sample search might be to get from

     start        finish

     2 1 3        1 2 3
     5 4 6        4 5 6
     7 8 .        7 8 .

where the "." is the empty square; the two possible first moves
slide either the 6 or the 8 to bottom right corner.

Is one sort of search better than the other for this problem?
```

I've implemented both versions of the search in Hot Cocoa Lisp. Breadth first search is the clear winner finding an efficient solution in under 2 seconds:

```
$ cd sliding_blocks
$ time node breadth_first.js
solution found! [5,2,1,4,3,0,1,2,5,4,3,0,1,4,5,8]
node breadth_first.js  1.76s user 0.11s system 99% cpu 1.883 total
```

The depth first version took longer than I wanted to wait but I gave it the simpler position

```
. 5 2
1 4 3
7 8 6
```

which can quickly be solved with [3,4,1,2,5,8] and got the result:

```
$ node depth_first.js
solution found! [3,6,7,8,5,4,7,8,5,4,7,8,5,4,3,6,7,8,5,4,7,8,
                 5,4,7,8,5,4,3,6,7,8,5,4,7,8,5,4,7,8,5,4,1,2,5,8]
```

The algorithm is quite simple and I've abstracted it out into *sliding_ blocks/search.hcl*.
A simple test of my sliding block puzzle api can be found in *sliding_ blocks/manual_ solution.hcl*:

```
$ node manual_solution.js
2 1 3
5 4 6
7 8 .

1 2 3
4 5 6
7 8 .

2 1 3
5 4 6
7 8 .
```

Question 3:

```
Explain in your own words what exactly is meant by "P" and "NP" as
complexity classes in computer science, why this is such an important
question, and what is and isn't known about them. Give explicit
examples of problems that are in each of these classes, and explain
```

```
why the known algorithms have behaviors consistent with your P and NP
descriptions. You may use external sources if you need to - and if so
be clear which ones you used, of course - but the point here is to
convey to us your understanding, not to just summarize a wikipedia
article.
```

In complexity theory, P stands for polynomial and refers to the class of problems which can be solved within polynomial time. NP stands from non-deterministically polynomial and refers to the class of problems for which a given solution to the problem can be verified in polynomial time. The question of whether or not $P = NP$ is simply that of whether or not all non-deterministically polynomial problems can be solved in polynomial time. It has been proven that that if any of the class of NP problems are in P then they all are and it is typically conjectured that $P \neq NP$. There are many problems of practical significance that are known to be in NP but for which there are no known algorithms for solving them in polynomial time. If it were shown that $P = NP$ it would mean that we have much more efficient solutions than we have so far found to many problems that have been looked at extensively. Two examples of NP problems follow.

The hamiltonian cycle problem
Given a graph determine whether there exists a path that starts a given node and by following edges touches each node in the graph exactly once before returning to the original node. This problem is clearly in NP because given a solution to the problem one need only traverse the given path once (in $O(n)$ time) to determine whether it is in-fact a solution. A simple algorithm for finding such a solution is a backtracking search which follows edges looking for a valid path. Since each node could in principle have as many as $n - 1$ edges this search has a branching ratio of order n and a depth of n thus the algorithm runs in $O(n^n)$ time. An even simpler algorithm would be to enumerate the permutations of the nodes and then chech whether a path exists for that ordering. This would run in $O(n!)$ time.

The boolean satisfiability problem Given a formula involving ands, ors, nots, parentheses, and variables that contain unspecified boolean values, determine whether an possible assignement of true or false to each of the variables exists such that the formula evaluates to true. The problem is in NP because given an efficient boolean logic system a solution can be verified in a single pass. The simplest algorithm for finding a solution is probably to simply enumerate the possible true/false assignements which are isomorphic

to the powerset of the set of variables and thus requires $O(2^n)$ time.

I used wikipedia to refresh my memory about these two problems.

Question 4:

> `Explain what a "hash table" is, and what it's O() behavior looks like. Implement one and use it to make a histogram of a word counts in a large text file. The details (collision algorithm, hash function, programming language) are up to you.`

A hash table is a particular implementation of a key value store data structure (i.e. python's dictionaries, php's associative arrays, or javascript's objects). The hash table makes use of a hash function which is a function that is designed to deterministically map keys to seemingly random indices. The hash function should have the property that any two similar keys map to different indices. The hash table simply stores key/value pairs in a sparse array at the index determined by the hash function. Because the hash function is deterministic retreiving the key value pair is as simple as running the key throught the hash function and jumping to the resulting index. There is no need to look through the entire list for the pair we need. This meens hash tables make read and write operations $O(1)$ with respect to the number of elements in the hash.

I've implemented a hash table in C. It handles hash collisions by keeping key/value pairs in "buckets" which form linked lists at each index. To find a given pair, my program hashes the given key and finds the associated index then follows the linked list it finds there until it finds a "bucket" with the given key of reaches a null bucket (which would mean the pair is not in the hash). Because I'm statically allocating only 256 buckets (for the simple reason that my hash function is particularly straight-forward with one byte indices) read/write operations should be $O(1)$ when there are around 256 keys and with larger numbers of keys the complexity will approach $O(n)$ for a linked list divided by a constant factor of 256. For some reason on my MacBook, I get segfaults when I try to read files over a certain size in C so I'm just using the first chapter of Moby Dick:

```
$ cd hash
$ gcc -Wall words.c -o words && ./words moby_ch1
```

I've also made general test of the hash table api that can be run with:

```
$ gcc -Wall test.c -o test && ./test
```

Question 5:

> Discuss the the connections between and ideas behind between a lossless compression algorithm (your choice which) and information entropy. Using a large text file (perhaps the same one from the previous problem), calculate an approximation to the information entropy. Find how much that file can be compressed, using a standard compression tool (or one you've implemented, but that's not required) and discuss how that is related to the entropy. Repeat for a file of random text, and explain how the those results compare.

Information entropy is basically a measure of how random a set of data appears to be. Lossless compression relies on patterns in a file which can be represented more succinctly for example a series of fifteen xs can be represented by some encoding of the number 15 and the character x. If a file has the maximum amount of entropy for a file of its size then, by definition, it contains no patterns and cannot be compressed. The less entropy the file contains the more it can be compressed. Shannon's definition of entropy uses a number between 0 and 1 which should be equal to the optimal theoretical compression ratio.

I've generated a file of random bytes of the same size as the first chapter of Moby Dick and I've created a short script in ruby to calculate the entropy of both files:

```
$ cd entropy
$ ruby entropy.rb moby_ch1
0.55706724002445

$ ruby entropy.rb random
0.998049624345567
```

I refreshed my memory of the formula for entropy using the discussion at: http://stackoverflow.com/questions/990477/how-to-calculate-the-entropy-of-a-file.

I also used gzip to determine the practical compression ratios of the files:

```
$ ls -l
...
-rw-r--r--  1 olleicua  staff  12243 Feb 25 20:57 moby_ch1
-rw-r--r--  1 olleicua  staff  12243 Feb 25 21:02 random

$ gzip moby_ch1

$ gzip random

$ ls -l
...
-rw-r--r--  1 olleicua  staff   5893 Feb 25 20:57 moby_ch1.gz
-rw-r--r--  1 olleicua  staff  12278 Feb 25 21:02 random.gz

$ ruby -e "p 5893.0 / 12243.0"
0.481336273789104

$ ruby -e "p 12278.0 / 12243.0"
1.00285877644368
```

The observed compression ratio for the first chapter of Moby Dick is probably lower for the simple reason that gzip looks for patterns spanning multiple characters and my entropy calulation only looke at single characters. A more thorough calculation would have also looked at groupings of consecutive characters of sizes up to the size of the file. The compression ratio for the random text is probably greater than on for the simple reason that gzip was unable to compress it at all but did add headers to specify things like file length, type of encoding used, and checksum.