Comparison of Optimal Sorts

Robert McCartney

1. ALGORITHMS

The algorithms implemented for this assignment were heapsort, mergesort, quicksort, and introspective sort, a modification of quicksort invented by David Musser in [1]. Heapsort is an in-place sort that does not require extra memory space beyond the original data structure in order to sort the array. It also has a worst-case complexity of $O(n * logn)$, but in practice is often found to be slower than other sorts. Mergesort is another sorting algorithm with a worst-case complexity of $O(n * logn)$, and is furthermore a stable sort whereby two elements that are equal will preserve their relative order to each other while modifying their absolute positions within the array. However, the downside of mergesort is that it requires extra memory to sort the array, up to O(n) extra space in my implementation. With extremely large datasets, this can present a problem, and can also slow the sort through memory access times. Quicksort is often faster in practice than these two sorts, depending on the type and distribution of the data used, and is an in-place sort that does not require secondary data structures. Its localized and sequential method of sorting elements in adjacent memory locations also works well with a memory cache. The downside is that its average complexity of $O(n * logn)$ is significantly worsened by a lot of repeated values, which will reduce the effectiveness of splitting around the pivot. In the worst case, the algorithm becomes $O(n^2)$. The fourth algorithm, introspective sort, tries to combine the benefits of quicksort with the optimal worst-case performance of $O(n * logn)$. It does this by beginning with quicksort but switching to heapsort once the recursive depth reaches a cutoff. In my implementation this cutoff is defined as $2 * logn$. It also includes some other optimizations to speed up the sorting process. First, the pivot is selected as the median of the first, last, and middle elements of the current array. This prevents extremely bad pivots from being selected, but is vulnerable to attacks called "median-of-3 killer" where a dataset is constructed by the attacker with knowledge of our pivot choices to force the algorithm to use n^2 comparisons. However, the switch to heapsort at the cutoff depth prevents this attack from becoming a denial of service. A second optimization is that if the array is less than size 16 it switches to the iterative in-place insertion sort in order to finish the job. Even though insertion sort is $O(n^2)$ in the worst case, with small sized arrays it is a lot faster in practice since it does not require the system overhead that comes with recursive function calls. Finally, if the array is of size 2 or 3, introspective sort uses an in-place swap rather than a recursive call. This improved version of quicksort is in fact similar to what is used in Microsoft's .NET framework. The pseudo-code for this sorting algorithm is as follows:

```
//Implementation uses medians of medians
choosePivot(array, left, right):
  middle = (right-left)/2 + left
  return median(left, middle, right)

introSort(array, left, right, depthlimit):
  long size = right-left+1
  if size <= 16:
    if size == 0 || size == 1:
      //base case
      return
    elif size == 2:
      //optimization
      if array[left] > array[right]:
        swap(array, left, right)
    elif size == 3:
      //optimization
      sort3(array, left, right)
    else:
      //optimization
      insertSort(array, left, size)
  //quicksort until we reach depthlimit
  //then switch to heapsort
  else:
    if depthlimit == 0:
      heapSort(array, left, size)
    else:
      depthlimit = depthlimit - 1
      pivot = choosePivot(array, left, right)
      //partition function same as quicksort
      newPos = partition(array,left,right,pivot)
      introSort(array,left,newPos-1,depthlimit)
      introSort(array,newPos+1,right,depthlimit)
```

2. PROGRAMMING LANGUAGE AND DATA STRUCTURE

Copyright 2014 ACM X-XXXXX-XX-X/XX/XX ...$10.00.

In my implementation I used C++ for the sorting algorithms because of its speed and low-level functionality. I also wanted to use the good support C++ has for generic programming. Using templates, I implemented all of the sorting algorithms generically, sorting an array of 'typename T'. Thus, they can be used to sort any comparable type in C++, such as int, double, char, string, or float. In fact, because C++ allows classes to overload operators, my sorting algorithms can be used on any object that implements the 'operator>' function in its class definition. Despite these benefits of using C++, there was also a downside to the language, as C++ has a random package that leaves much to be desired.[1] Knowing this limitation, I switched over to Python, an excellent language for data and statistical applications, in order to sample from the four required distributions and then pass this data as text files to the C++ executable. Note that this passing of the data from creation in Python to sorting in C++ is completely automated for every size (1K, 10K, 50K, 500K, and 1 million) and every distribution (ordered, quarter-sorted, random, and Poisson), using Python's *subprocess* function call. The end result was an implementation that was fast, generic, and extensible, able to shift between underlying data types and sizes with ease.

3. RESULTS

Each of the four sorting algorithms was run on input size n ranging from 1K to 1 million, and at each size the data was either random, sorted for the first 25% of the data and then unsorted from there, completed ordered already, or sampled from the Poisson distribution with parameter λ equal to $\frac{n}{2}$. For each of these various datasets, each sorting algorithm was rerun 10 times in order to determine average execution time. This was done for input data of both ints and doubles.[2] The results can be seen in Figures 13 and 14. Note that execution time was not significantly impacted by the change from 4 byte int to 8 byte double. All graphs that follow will be for real-valued doubles. See Figures 1 through 3 for graphs of the time results for each of the fours sorts on the different distribution types (note that random and quarter-sorted graphs are nearly identical so the quarter-sorted graph is omitted here).

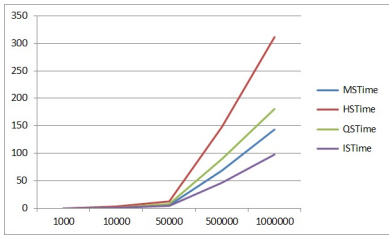

Figure 1: Ordered: Time (ms) vs Input Size

In ordered and random distributions, heapsort performed the worst as was predicted above. The intro sort proved to be the fastest, especially for an ordered distribution where its switch to insertion sort at size 16 would need to do no swaps at all and would run at O(n). In the Poisson distribution, it is clear that quicksort struggled a lot with the many repeated values clustered around the mean. This caused it to deviate significantly from O($n * logn$). While intro sort was no longer faster than mergesort, it still beat heapsort and represented a significant improvement in the quicksort algorithm. Looking at Figures 4 through 6 you can see the number of comparisons used by each algorithm.

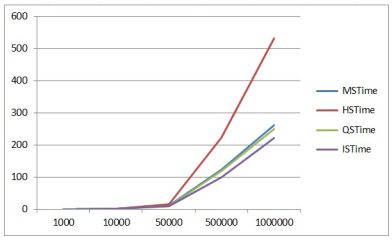

Figure 2: Random: Time (ms) vs Input Size

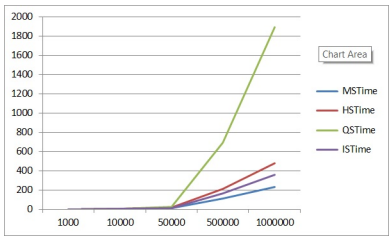

Figure 3: Poisson: Time (ms) vs Input Size

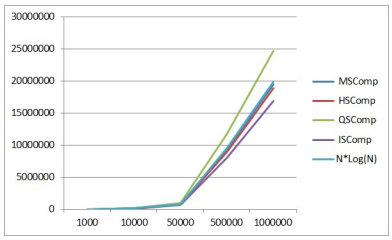

Figure 4: Ordered: Comparisons vs Input Size

In ordered, random, and quarter-sorted[3], they closely follow the plotted line of $y = n * logn$. In Poisson, the number of quicksort comparisons jumps up, closer to an exponential path, while intro sort increases its number of comparisons only linearly. Looking at Figures 7 through 9 you can see the number of recursive calls used by quicksort and intro sort for each type of distribution. Clearly, intro sort made significant improvements in the number of function calls needed to sort every array size.

As another way to look at the data, I have shown in Figures 10 through 12 each of the sorting algorithms by themselves, with each type of distribution graphed as the time in milliseconds needed to sort that distribution ver-

[1]It has been upgraded in C++11, but that standard is not available in the computer labs yet.

[2]Note the exception that a variable that is Poisson distributed only takes on integer values.

[3]Again not shown due to redundancy

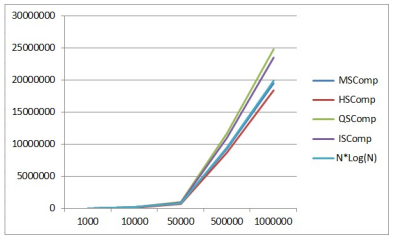

Figure 5: Random: Comparisons vs Input Size

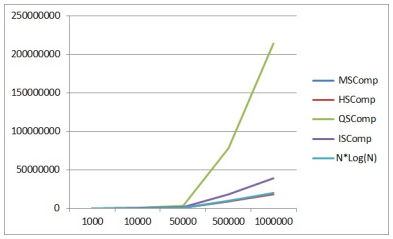

Figure 6: Poisson: Comparisons vs Input Size

sus the data input size.[4] Here, it is clear that mergesort and heapsort work best on ordered distributions, and are slowest on random distributions. Quicksort and intro sort are also fastest on ordered distributions, but are susceptible to slowing down when given repeated values tightly grouped around the mean, like in the Poisson distribution. Of course as discussed above, intro sort's reduction in efficiency is mild compared to the significant reduction seen in traditional quicksort.

[4]Graph for heapsort is very similar to mergesort so it is omitted here.

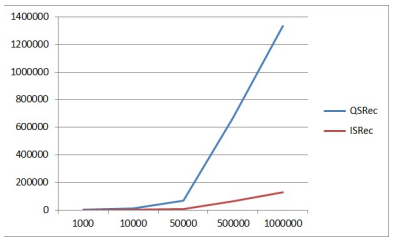

Figure 7: Ordered: Recursive calls vs Input Size

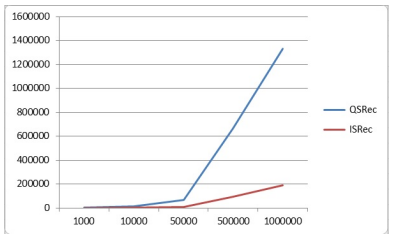

Figure 8: Random: Recursive calls vs Input Size

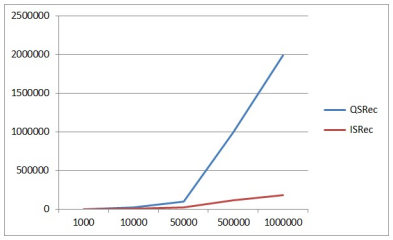

Figure 9: Poisson: Recursive calls vs Input Size

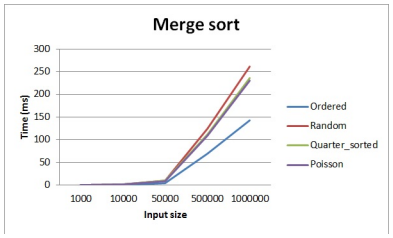

Figure 10: Sort speed by distribution

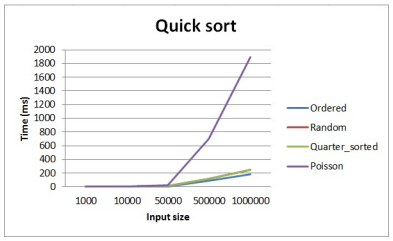

Figure 11: Sort speed by distribution

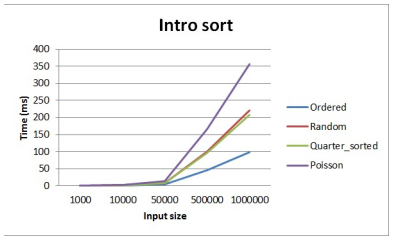

Figure 12: Sort speed by distribution

Dist	Size	MSTime	MSComp	HSTime	HSComp	QSTime	QSComp	QSRec	ISTime	ISComp	ISRec
Ordered	1,000	0	9,488	0	8,813	0	11,127	1,333	0	6,942	127
Random	1,000	0	9,488	0	8,436	0	10,760	1,328	0	10,761	179
Quarter	1,000	0	9,488	0	8,458	0	11,041	1,343	0	10,920	183
Poisson	1,000	0	9,488	0	8,429	0	14,218	1,806	0	9,837	287
Ordered	10,000	1	129,520	3	122,288	1	155,020	13,325	1	108,986	2,047
Random	10,000	2	129,520	2	117,673	2	157,389	13,327	2	145,951	1,907
Quarter	10,000	2	129,520	2	118,501	1	156,203	13,327	1	150,494	1,895
Poisson	10,000	2	129,520	2	117,741	3	300,505	19,222	2	220,004	6,621
Ordered	50,000	5	767,232	12	727,738	8	963,490	66,722	4	645,916	8,191
Random	50,000	11	767,232	15	704,874	10	942,034	66,659	9	900,071	9,535
Quarter	50,000	10	767,232	14	710,018	10	948,696	66,687	8	863,137	9,611
Poisson	50,000	9	767,232	15	704,895	27	2,862,505	98,098	14	1,445,977	23,829
Ordered	500,000	69	9,237,856	148	8,922,789	90	11,728,165	666,789	46	7,967,247	65,535
Random	500,000	124	9,237,856	224	8,698,341	120	11,717,068	666,740	100	10,948,837	95,421
Quarter	500,000	112	9,237,856	214	8,756,538	116	11,646,224	666,774	97	11,004,675	95,707
Poisson	500,000	109	9,237,856	213	8,698,163	695	77,807,244	993,269	166	18,434,759	111,023
Ordered	1,000,000	143	19,475,712	311	18,864,660	181	24,734,166	1,333,251	98	16,934,480	131,071
Random	1,000,000	262	19,475,712	531	18,397,212	250	24,749,817	1,333,351	221	23,488,561	190,451
Quarter	1,000,000	236	19,475,712	520	18,374,804	246	24,981,832	1,333,354	207	23,118,366	191,203
Poisson	1,000,000	230	19,475,712	480	18,398,034	1,896	214,373,927	1,990,166	356	39,132,588	179,211

Figure 13: Dataset for 4 byte int input

Dist	Size	MSTime	MSComp	HSTime	HSComp	QSTime	QSComp	QSRec	ISTime	ISComp	ISRec
Ordered	1,000	0	9,488	0	8,813	0	11,349	1,345	0	6,942	127
Random	1,000	0	9,488	0	8,421	0	11,120	1,336	1	10,748	191
Quarter	1,000	0	9,488	0	8,515	0	10,983	1,340	1	10,362	179
Poisson	1,000	0	9,488	0	8,418	0	14,096	1,805	0	9,567	277
Ordered	10,000	1	129,520	2	122,288	1	152,828	13,345	1	108,986	2,047
Random	10,000	2	129,520	2	117,662	2	154,963	13,346	1	150,969	1,871
Quarter	10,000	2	129,520	2	118,859	2	153,987	13,341	1	146,359	1,917
Poisson	10,000	1	129,520	2	117,742	3	306,231	19,228	3	219,077	6,587
Ordered	50,000	6	767,232	11	727,741	7	941,718	66,745	4	645,916	8,191
Random	50,000	10	767,232	13	705,009	9	934,443	66,710	7	872,792	9,547
Quarter	50,000	9	767,232	13	702,584	8	949,803	66,679	7	879,538	9,565
Poisson	50,000	9	767,232	12	704,884	27	2,828,622	98,093	14	1,451,365	23,583
Ordered	500,000	60	9,237,856	131	8,920,674	89	11,939,192	666,579	48	7,967,247	65,535
Random	500,000	103	9,237,856	160	8,698,013	111	11,691,459	666,777	91	10,711,145	95,281
Quarter	500,000	93	9,237,856	156	8,714,663	109	11,887,688	666,802	91	10,991,882	95,737
Poisson	500,000	91	9,237,856	154	8,698,873	710	77,850,176	993,246	165	18,391,365	115,897
Ordered	1,000,000	124	19,475,712	275	18,865,722	177	24,793,487	1,332,992	99	16,934,480	131,071
Random	1,000,000	214	19,475,712	346	18,396,851	231	24,712,359	1,333,410	191	22,720,485	190,521
Quarter	1,000,000	194	19,475,712	331	18,339,276	225	24,789,998	1,333,388	185	22,721,027	191,649
Poisson	1,000,000	193	19,475,712	326	18,396,808	1,948	214,462,872	1,990,162	353	39,146,813	179,669

Figure 14: Dataset for 8 byte double input

4. REFERENCES

[1] D. R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience (Wiley)*, 27(8):983–993, 1997.