Algorithm Analysis
Project

Ege University Computer Engineering

Özlem GÜRSES 05-07-8496
Savaş YILDIZ 05-07-8569
Umut BENZER 05-06-7670

0.1
January 3rd, 2010

Table of Contents

Introduction

Our program implements Bubble Sort, Quick Sort sorting algorithms in Part 1. It also implements Exhaustive Search and Dynamic Programming approaches to solve the Knapsack problem in Part 2.

Our program is written in Java and JRE 6 is required to run it.

We used NetBeans 6.9.1 as our IDE.

Some Important Notes…

Please note that, running times returned from our program is in nanosecond instead of milliseconds. Since for some inputs, algorithm works faster than 1ms, we thought it would be better to use nano-seconds to measure the running times. *(1 Nanoseconds = 0.000001 Milliseconds)*

Please also note that, our program doesn't measure running time by simply running algorithm for just one time. We all use multi-tasking operating systems. We realized that, every time we run the algorithms, different and event not close running times are generated.

For example, when we run Bubble Sort once, it was completed in 10ms. With the same array, same computer, we tried again and it was completed in 55ms than! So, we run algorithm a lot of times and took the average. Even it wasn't enough but, with limited computational power of our computers and limited time, it was the best we could.

Sample count is changeable with parameters.

And a last point, we tried to use \t tab. However, in Java, \t added constant amount of blank into the console and didn't format the output. So we removed it (since it was unreadable), did some research and found our solution to format output to look good.

Using Our Program…

Our program has 4 parts: Part 1, Part 1 Benchmark, Part 2 and Part 2 Benchmark. In the following sections, we'll explain them all in detail.

Part 1

We choose **Bubble Sort** from Group 1 and **Quick Sort** from Group 2. We ran them and get results. The detailed results and plots can be found at Appendix A.

Our Evaluation

Our graphs have some unpredicted and unexpected noise in running times. We think these are caused by Windows® Operating System and other background process and random generated interesting arrays.

However, in general, we can clearly see that Quick Sort is far faster than Bubble Sort, especially in big inputs. We can also see that, both Quicksort and Bubble Sort are in range of efficiency classes we know.

Bubble Sort is a member of $O(n^2)$ in worst case, $O(n^2)$ in average and O(n) is best case. From the plot, we can clearly see that, Bubble Sort's performance is better than n^2 but worse than n. It is closer to n^2. It is consistent with its theoretical results both in numbers of key comparisons and running time.

Quick Sort is a member of $O(n^2)$ in worst case, O(n log n) in average and O (n log n) in best case. From the plot, we can clearly see that, Quick Sort's performance is far better than n^2 but, worse than n log n. *(We used 20 as multiplier)* However, it is closer to n log n than n^2. It is consistent with its theoretical results.

Running the Program

To run these algorithms in our program, use these parameters:

```
p1 inputFileName outputFileName [sampleCount]
```

`p1` indicates that program will run part 1.

`inputFileName` is the input file's name. Array counts which are wanted to be tested must be separated by space in the input text.

`outputFileName` is output file's name. The location must be writable.

`sampleCount` is optional. If it is not provided, program run each algorithm 1000 times and gets the average running time. It is recommended to use higher values if CPU permits. It is known that, background works and other programs affect running times incredibly, especially when sample count is low.

Examples

```
java -jar "C:\AlgorithmAnalysis.jar" p1 input_part1.txt output_part1.txt
java -jar "C:\AlgorithmAnalysis.jar" p1 input_part1.txt output_part1.txt 2000
```

Part 1 Benchmark

Part 1 Benchmark is a benchmark platform which runs the same algorithms with Part 1, but with automated input and different output styles and parameters. Values in Appendix A are taken by this benchmark platform.

With automated input, we can test lots of input faster and the output is designed to be copy/pasted easily to a plot generator program (in our case, Microsoft® Excel).

Running the Program

To run these algorithms in our program, use these parameters:

```
p1benchmark outputFileName sampleCount fromItem toItem stepSize
```

`p1benchmark` indicates that program will run part 1 in benchmark mode.

`outputFileName` is output file's name. The location must be writable.

`sampleCount` is the value, each algorithm will run for each input sets and get an average running times. It is a good thing, because background processes and other programs may affect running times of algorithms. It is known that, background works and other programs affect running times incredibly, especially when sample count is low.

`fromItem` is a numeric value, the starting value of array size.

`toItem` is a numeric value, the end value of array size.

`stepSize` is a numeric value, the increase number between `fromItem` to `toItem`.

Examples

```
java -jar "C:\AlgorithmAnalysis.jar" p1benchmark part1bench.txt 1000 0 10000 2000
```

Part 2

We ran both algorithms and get some results. The detailed results and plots can be found at Appendix B.

Our Evaluation

Our graphs have some unpredicted and unexpected noise in running times. We think these are caused by Windows® Operating System and other background process and random generated interesting arrays.

However, in general, we can clearly see that Dynamic Programming method is faster Exhaustive Search method. However, if there is low item count, (such as 10) and capacity is too big (such as 200) than Exhaustive Search may be faster in these extreme conditions.

For more information plots could be seen in Appendix B.

Exhaustive Search for Knapsack problem is a member of $O(2^n)$. As seen from plots, our results are about the same. (We used 3000 as multiplier) Dynamic Programming has a complexity of O(nk)n where n is item count and k is capacity. As seen from plots, it is about the as the theoretical results. (We used 50000 as multiplier.) Practical results are consistent with their theoretical results both in numbers of key comparisons and running time.

Running the Program

To run these algorithms in our program, use these parameters:

```
p2 inputFileName outputFileName [sampleCount]
```

`p2` indicates that program will run part 2.

`inputFileName` is the input file's name. It must be formatted as shown in project document.

`outputFileName` is output file's name. The location must be writable.

`sampleCount` is optional. If it is not provided, program run each algorithm 1000 times and gets the average running time. It is recommended to use higher values if CPU permits. It is known that, background works and other programs affect running times incredibly, especially when sample count is low.

Examples

```
java -jar "C:\AlgorithmAnalysis.jar" p2 input_part2.txt output_part2.txt
java -jar "C:\AlgorithmAnalysis.jar" p2 input_part2.txt output_part2.txt 2000
```

Part 2 Benchmark

Part 2 Benchmark is a benchmark platform which runs the same algorithms with Part 2, but with automated random generated input and different output styles and parameters. Values in Appendix B are taken by this benchmark platform.

With automated input, we can test lots of input faster and the output is designed to be copy/pasted easily to a plot generator program (in our case, Microsoft® Excel).

Running the Program

To run these algorithms in our program, use these parameters:

```
p2benchmark outputFileName sampleCount fromItem toItem fromCapacity toCapactiy
```

`p2benchmark` indicates that program will run part 2 in benchmark mode.

`outputFileName` is output file's name. The location must be writable.

`sampleCount` is the value, each algorithm will run for each input sets and get an average running times. It is a good thing, because background processes and other programs may affect running times of algorithms. It is known that, background works and other programs affect running times incredibly, especially when sample count is low.

`fromItem` is a numeric value, the starting value of item size.

`toItem` is a numeric value, the end value of item size. If this value is larger than 10, you will have serious problems with running time, since Exhaustive Search for Knapsacks is a member of $O(2^n)$

`fromCapacity` is a numeric value, the starting value of Knapsack capacity.

`toCapacity` is a numeric value, the end value of Knapsack capacity.

Examples

```
java -jar "C:\AlgorithmAnalysis.jar" p2benchmark part2bench.txt 1000 0 10 0 50
```

Benchmark Computer

Benchmarks are done under these conditions:

- AMD Turion X2 Dual-Core Mobile RM-74 2.20Ghz Multicore Processor
- 4GB of RAM
- Windows 7 Professional x86-64
- JRE 6 Update 23 x86-64
- On plug, High Performance Mode,
- No other applications are working.
- Sample sizes are indicated in tables.