Mark as done

Analysis of Algorithms

Introduction

The purpose of this practical is to understand the analysis of algorithms so they can be compared. Don't forget that you are required to submit the indicated exercise through myLearn for assessment.

Algorithms

An algorithm is a step-by-step process that can be followed. Useful algorithms typically take some input, perform some calculations on that input, and output some result. However, there can be more than one way to perform the same operation. Analysis of algorithms allows us to compare the resources (typically either the time taken or memory space required) so we can select the best algorithm for our circumstances.

Most of the time when we are analysing algorithms, we concentrate on the limit of how the algorithm performs as the input goes to infinity. While this is not always realistic, it does give us a useful comparison; it allows us to compare algorithms with arbitrarily large data. We also typically ignore minor details of the algorithms. For example, if an algorithm performs a single constant-time operation once at the beginning of processing, but then goes into a linear loop, the constant-time operation can be ignored. Practically, such an operation will take some time, but in the overall scheme of things, the effect of that one-time constant operation is negligible in comparison with the loop. Similarly, an algorithm that performs one linear loop followed by another may take twice as long, but we ignore constant differences.

Input also has an effect on some algorithms. For example, quicksort performs poorly on data that is already sorted. To overcome this limitation, we typically concentrate on worst-case performance of an algorithm. That is, assume it is given data that makes the algorithm take as much time or use as much resources as possible for that data size.

We can compare the run time or memory usage of different algorithms using Big O notation. With Big O notation, only the leading term of a polynomial matters; lower terms and coefficients are ignored. We can then class algorithms by determining the Big O value they have. An algorithm with a lower Big O class is more efficient than one in a larger Big O class.

While often harder to analyse, another important property of algorithms is their average-case performance. An algorithm can perform poorly in pathologically bad cases, but perform much better in the average case. For example, quicksort is O(n2), but in the average case runs in time O(n log n).

Exercises

Use material from this week's lectures to perform the following exercises:

1. **[This exercise should be submitted through the Assessments section of myLearn for Tutorial Exercise 8]** A farrier was contacted one Sunday and asked to shoe a horse. Because Sunday was his day off, the farrier advised the rider to come back tomorrow, and that it would cost $150. But the rider kept insisting that the horse needed to be shod immediately, until eventually the farrier wore down and made the following offer: he would shoe the horse for free, but the rider needed to buy the nails from him and would be charged $0.01 for the first nail, and twice the price of the previous nail for each nail after that (e.g. the second nail would cost $0.02, the third $0.04, and so on). Given that a horse needs four shoes, and each shoe needs six nails, complete the following program to determine the total cost for the rider. It just goes to show how quickly things can grow with exponential growth!

```
total_cost = 0
cost_of_nail = 1
for horse_foot in range(4):
    for nail in range(6):
        # TODO: Add price of current nail to total_cost and determine the cost of the next nail
        pass

print("The total cost to shoe the horse would be ${0:.2f}".format(total_cost / 100))
```

2. What is the order of growth of:
    1. $n^3 + n$
    2. $3n^3$
    3. $3n^3 + 100000n^2$
    4. $500000n^3$
3. What is the order of growth of $(n^2 + 5)(67n^3 + 93)$? Remember, you only need the leading term!
4. Implement functions named `linear` and `bisection`, each of which take a list and a target value as parameters. `linear` should perform a linear search, and `bisection` a bisection search. They should each return the index of the target value if it is in the list, or *None* if it is not.
5. Implement each of the search algorithms covered in lecture, add debugging information, and look through the output to understand how they work. Can you write unit tests to help ensure the sorts are all working?
6. Implement the hashtable covered in lecture, add debugging information, and look through the output to understand how the hashtable functions work. Can you write unit tests to ensure the functions are working correctly?

Last modified: Thursday, 1 February 2024, 10:50 AM