Mission 12: Sorting Things Out

Start date: 20 September 2017
Due: 3 October 2017, 23:59

Readings:

- None

Background

You heard from your friends that the IT staff used *merge sort* to sort the name list of all cadets at the Source Academy, before they could construct the *binary search tree*. You also remembered you have hearded about **quicksort** before.

"Emmm, it is called **quicksort**. So, literally, shouldn't it be very fast? I bet that I could restore the name list faster with it."

Some historical facts

Quicksort is an efficient sorting algorithm, sometimes also known as *partition-exchange sort*.

It was first developed in 1959 and published in 1961, by Tony Hoare, who is a British computer scientist and winner of the *Turing Award* in 1980. He was a visiting student at Moscow State University when he invented this algorithm. **Quicksort** was initially intended to be used to help translation between English and Russian.

> Turing Award is an annual prize given by the Association for Computing Machinery (ACM) to "*an individual selected for contributions of a technical nature made to the computing community*". The Turing Award is generally recognized as *the highest distinction in computer science* and the "*Nobel Prize of computing*". Click here (https://en.wikipedia.org/wiki/Turing_Award) for more information.

This mission has **three** tasks. Please remember to press "*Save*" button before you submit this mission.

Task 1:

Quicksort is considered to be:

- Easy to understand;
- Moderately not simple to implement correctly;

- A little hard to analyze;
- Challenging to further optimize.

However, with the magic power of Source, **quicksort** can be easily understood and implemented. In order to implement quicksort using the list library in Source, we first need to implement a `partition` operation on a list that adheres to the following specification.

The function `partition` is applied to an possibly unsorted list `xs` and an element `p` that we call *pivot*. It returns a pair of two possibly unsorted lists, that when combined contain all elements of `xs`. The head of the pair contains all elements of `xs` that are smaller than or identical to `p` and the tail of the pair contains all elements of `xs` that are larger than `p`. In this task, it is irrelevant in what order the elements of `xs` appear in the result lists.

For example, `partition(list(5,6,1,7,9,2), 6)` may return a result that is equal to `pair(list(2, 1, 6, 5), list(9, 7))`. The order does not matter. A result that is equal to `pair(list(5, 6, 1, 2), list(7, 9))` is also allowed.

Please fill in the template given and implement the `partition` operation on a list.

Task 2:

Quicksort relies on the `partition` operation to sort things out. The cases of sorting the empty list and one-element lists are not difficult, as we hope you see easily. For other lists, we choose the `head` as *pivot*, and split the `tail` using `partition`. Since `partition` does not sort the resulting lists, we sort them using quicksort itself (using wishful thinking), before we combine them into the sorted overall result.

Use the template given to implement your **quicksort**. *Notice*: you need to copy-and-paste the `partition` function from Task 1 to here and make use of it.

Task 3:

When implemented well, **quicksort** can be about two or three times faster than its main competitors, *merge sort* and *heap sort*. Thus, it is still commonly used in practice till now.

Thus, it becomes essential for us to understand its performance here.

Please answer the following questions:

1. Find out the order of growth in time for `partition`. Also, explain whether it gives rise to an iterative process or a recursive process.
2. Find out the order of growth in time for `quicksort` under the following three scenarios:
   a) The given list is in ascending order, like `list(1, 2, 3, ..., n)`;
   b) The given list is in descending order, like `list(n, n - 1, ..., 1)`;
   c) Find a list with the numbers 1 to 23 that can be sorted in the fastest way using **quicksort**. Let us assume such best cases exist for any `n`. Can you guess the order of growth for sorting such best case scenarios, using Big-Theta notation?
3. Based on your answer to the last question, comment on the performance of `quicksort`.

Submission

Submit your mission on the Source Academy.

IMPORTANT: Make sure that everything for your programs to work is on the left hand side (Editor) and not in the Side Content! This is because only the programs in the Editor are preserved when your Avenger grades your submission.