Searching. Sorting.

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Let's define the search problem

- Data are available in the internal memory, as a sequence of records $(k_1, k_2, ..., k_n)$
- We search for the record that has a certain value for one of its fields, called the **search key**.
- We return the position of the record in the given sequence, or -1 if the **search key** was not found
- Depending on the structure of the sequence of records, we distinguish two possibilities:
  - Searching with unordered keys
  - Searching with ordered keys

Searching with unordered keys

Problem specification

- **Data**: A is a sequence of records with n elements, where $n \in \mathbb{N}$.
- **Result**: *index*, where $(0 \leq index \leq n-1)$, the search key's position in sequence A, or $index = -1$, if the key was not found.

Searching with unordered keys

Sequential search (iterative and recursive)

Examine the source code in **ex08_sequential_search.py**

What's the computational complexity:

- Best case: search key is the first element checked, so $T(n) = 1 \in \Theta(1)$
- Worst case: search key is not found, so $T(n) = n \in \Theta(n)$

Searching with unordered keys

What's the computational complexity:

- Average case: let's assume[1] 50% of searches end with the element not found, and when the element is found, it has the same probability of being on any position in the list. This time, we simulate $2*n$ runs; during n runs, the element is not found, and during the remaining runs, it is found on each of the list's n positions;

$$T(n) = \frac{\overbrace{1+2+...+n}^{key\ \ found,\ n\ \ searches}}{2*n} + \frac{\overbrace{n+n+...+n}^{key\ \ not\ \ found,\ n\ \ searches}}{2*n} \in \Theta(n)$$

[1] an assumption so that we may provide probability P(I)

Searching with unordered keys

Sequential search (iterative and recursive)

Examine the source code in **ex08_sequential_search.py**

What about a recursive implementation[2]?

$$\begin{cases} T(n) = 1, n = 1 \\ T(n) = T(n-1) + 1, n \geq 1 \end{cases}$$

Leads to a telescopic sum that resolves to $T(n) \in Theta(n)$

[2]Only for fun ☺, as the stack usually frowns upon this implementation

Searching with ordered keys

Problem specification

- **Data**: A is a sequence of records with n elements, where $n \in \mathbb{N}$ so that $\forall 0 \leq i < j \leq n-1, A[i] <= A[j]$
- **Result**: *index*, where $(0 \leq index \leq n-1)$, the search key's position in sequence A, or $index = -1$, if the key was not found.

Searching with ordered keys

Binary search (iterative and recursive)

Examine the source code in **ex09_binary_search.py**. Also pay attention to the *test_binary_search* method.

What's the computational complexity:

- Best case: search key is the first element checked, so $T(n) = 1 \in \Theta(1)$
- Worst case: search key is not found. $\begin{cases} T(n) = 1, n = 1 \\ T(n) = T(n/2) + 1, n \geq 1 \end{cases}$, each time halving the array in $O(1)$ time. Since we can do this $log_2 n$ times, the complexity in the average and worst cases is $\Theta(log_2 n)$

The sorting problem

Sorting

Rearrange a data collection in such a way that the elements of the collection verify a given order.

Problem specification

- **Data**: A is a sequence of records with n elements, where $A = (a_1, a_2, ..., a_n), a_i \in \mathbb{R}, i = 1, n$
- **Results**: A', where A' is a permutation of A, having sorted elements: $a'_1 \leq a'_2 \leq ... \leq a'_n$.

The sorting problem

Terminology:

- **Internal sort** - data to be sorted are available in the internal memory
- **Streaming sort** - data becomes available sequentially (being read from the network, a database etc.)
- **In-place sort** - transforms the input data into the output, only using a small additional space. Its opposite is called out-of-place.
- **Sorting stability** - the original order of multiple records having the same key is preserved
- **Adaptive sort** - an algorithm that takes data structure into account while sorting
- **Hybrid sort** - an algorithm that uses two or more sorting algorithms to speed up sorting

Demo

Stable sort example

Examine the source code in **ex10_stable_sort.py**

Sorting algorithms

A few algorithms that we will study:

- (Binary) Insertion sort
- Quick Sort
- Merge sort
- TimSort

Insertion sort

- Traverse the list once, examining each element a single time
- Insert the examined element at the right position in the subsequence of already sorted elements.
- The sub-sequence containing the already processed elements is kept sorted, so that, at the end of the traversal, the entire sequence is sorted.
- Very good for short lists, or lists that are nearly sorted (with only a few elements out of place)

Insertion sort – fact sheet

In-place	✓	Stable	✓
Streaming	✓	Space	$O(1)$
Time	**Best** $O(n)$	**Average** $O(n^2)$	**Worst** $O(n^2)$

Binary Insertion sort

- An optimization of insertion sort
- Since the sequence of already processed elements is sorted, we use binary search to find the place of each new element

Binary Insertion sort

Question

What happens if use (binary) insertion sort to sort **linked lists** instead of **arrays**? What are the differences?

Binary) Insertion sort

Example

Examine the source code in **ex11_insertion_sort.py**

Binary Insertion sort – fact sheet

In-place	✓	Stable	✓
Streaming	✓	Space	$O(1)$
Time	**Best**	**Average**	**Worst**
	$O(n * log_2 n)$	$O(n^2)$	$O(n^2)$

Quick Sort

Based on the *divide and conquer* technique

1. **Divide:** partition array into 2 sub-arrays such that elements in the lower part $\leq$ elements in the higher part.

Partitioning

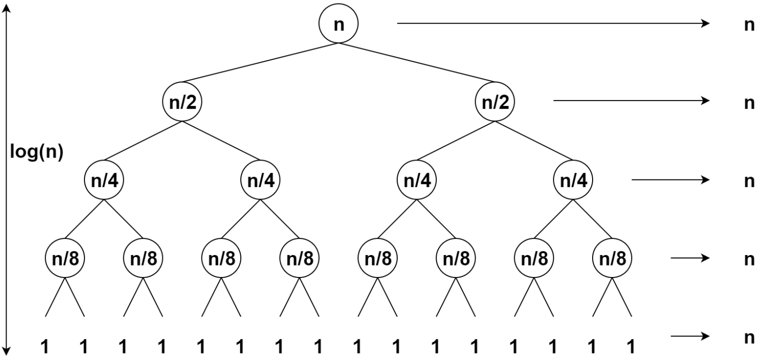
Re-arrange the elements so that the element called pivot occupies the final position in the sub-sequence. If i is that position: $k_j \leq k_i \leq k_l$, for $Left \leq j < i < l \leq Right$

2. **Conquer:** recursively sort the 2 sub-arrays.
3. **Combine:** trivial since sorting is done in place.

Quick Sort - time complexity

- An example of in-place, divide & conquer, non-stable sort ☺
- Its time complexity depends on the distribution of splits, namely pivot selection. Some strategies:
  - Choose the first or last element
  - Choose a random element, or the one in the middle
  - Median of three elements in the partition (e.g., first, midpoint, last)

Quick Sort - best partitioning

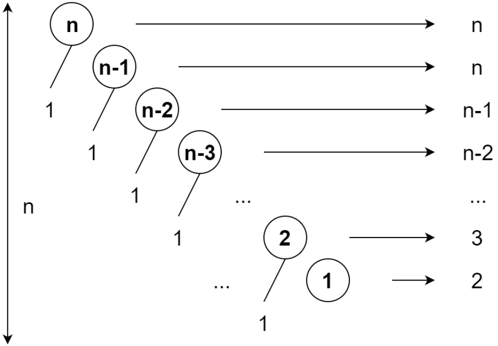


- We partition n elements $\log_2 n$ times, so $T(n) \in \Theta(n \log_2 n)$

Quick Sort - worst partitioning

- In the worst case, function Partition splits the array such that one side of the partition has only one element: $T(n) = T(1) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) \in \Theta(n^2)$

Quick Sort - Worst case



- Worst case appears when the input array is sorted or reverse sorted, and we select the first or last element as the pivot
- The n elements are partitioned n times, $T(n) \in \Theta(n^2)$

Quick Sort – fact sheet

In-place	✓	Stable	**X**
Streaming	**X**	Space	$O(log_2 n)$
Time	**Best** $O(n * log_2 n)$	**Average** $O(n * log_2 n)$	**Worst** $O(n^2)$

Merge Sort

- Based on the divide & conquer approach.
- The list to be sorted is divided in two sub-lists that are sorted separately. The sorted sub-lists are then merged.
- Each sub-list is sorted using the same approach until we get to sub-lists of length 1, which we know are sorted.

Merge Sort

+ Merge sort has good time complexity and it's easy to write an implementation that works well...
– It has non-constant extra-space complexity
– Time is wasted when merging very short lists (length 1, 2, 4, ...)

Merge Sort – fact sheet

In-place	**X**	Stable	**X**
Streaming	**X**	Space	$O(n)$
Time	**Best**	**Average**	**Worst**
	$O(n * log_2 n)$	$O(n * log_2 n)$	$O(n * log_2 n)$

Sorting Algorithm Optimization

- There's no really "best" sorting algorithms, as data size, structure, cost of comparisons, CPU cache sizes and other things influence actual run times
- e.g., merge-insertion sort (a.k.a Ford-Johnson algorithm) was known for a long time as the sorting algorithm with the smallest number of element comparisons
- Combine the strengths of several algorithms into **hybrid algorithms** – merge or quick sort that default to insertion sort for short lists

Sorting Algorithm Optimization

Sorting algorithms visualized

`https://www.toptal.com/developers/sorting-algorithms`

TimSort

- Developed for Python in 2002 by Tim Peters
- Adapted in Java 7, Android, Swift, V8 and Rust
- A hybrid algorithm built on merge sort that employs binary insertion sort for small sections of the list, leverages natural element ordering and in-cache data ☺

A Word of Warning

You'll find **many** resources on the interwebs showing "TimSort", but most of them are either incomplete, or straightforward combinations of merge and insert sort. Some of them are even buggy ☹

TimSort

The Real TimSort

- The definitive description of the algorithm by its author (`https://svn.python.org/projects/python/trunk/Objects/listsort.txt`) together with its source code (`https://github.com/python/cpython/blob/main/Objects/listobject.c`)
- Some more details regarding it (`https://mail.python.org/pipermail/python-dev/2002-July/026837.html`)

TimSort

A few other resources for those interested

- Good (video) descriptions (`https://www.awesomealgorithms.com/home/tim-sort`) and (`https://ericmervin.medium.com/what-is-timsort-76173b49bd16`)
- Detailed, step by step description **with Python source code** (`https://vladris.com/blog/2021/12/30/timsort.html`)
- Source code in Python from the RPython project (`https://github.com/reingart/pypy/blob/master/rpython/rlib/listsort.py`)

TimSort

In the author's own words...

This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it ☺). It has supernatural performance on many kinds of partially ordered arrays (less than lg(N!) comparisons needed, and as few as N-1), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.
In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs "intelligently". Everything else is complication for speed, and some hard-won measure of memory efficiency.

`https://svn.python.org/projects/python/trunk/Objects/listsort.txt`

TimSort

Most existing lists include sublists that are already sorted (e.g., [1, 4, 5, 2, 99, 6, 9, 11, 10, 90] has [1, 4, 5] and [6, 9, 11] length-3 sublists)

1. Determine the value for *minrun*, a parameter that balances good insert sort performance ($32 < minrun < 64$) so that the list is divided into a power of 2 runs
2. Sort consecutive runs of length *minrun* using binary insertion sort
3. Place runs on a stack ($O(n)$ space complexity)
4. Merge consecutive runs from the stack using merge sort (this leads to a stable sort)

TimSort - optimizations

- While elements are sorted, keep going even after *minrun*
- A run sorted in descending order can be reversed to ascending in $O(n)$ (check order of first two elements)
- Only call insertion sort for those parts of the run that are not already sorted
- Merging often (and soon after runs are completed) raises the chance that data is (still) in the CPU cache
- Introduce a stack invariant on the length of the merges to ensure balanced merges
- Optimize merging by finding which elements remain in the existing order (e.g., best case when the first element of run A is larger than the last of run B)
- When merging uses data from the same run repeatedly, use an exponential search (galloping) to identify how many more elements can be used

TimSort – fact sheet

In-place	✓	Stable	✓
Streaming	**X**	Space	$O(n)$
Time	**Best**	**Average**	**Worst**
	$O(n)$	$O(n * log_2 n)$	$O(n * log_2 n)$