COMP6700/2140 Sorting and Searching

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

May 2017

Topics

1. Recursive Algorithms: Divide and Conquer
2. Quick Sort
3. Merge Sort
4. Tim Sort — A Hybrid Algorithm
5. Binary Search
    - 2006 Binary Search Break: Abstraction Leaked ☹

Divide and Conquer Strategy

Computational problems are difficult, often very difficult. When it is possible to reduce the problem size and apply the same solving strategy (either divide again, or deal with an ultimately reduced problem, which is usually trivial), one talks about **divide and conquer** algorithms.

- When only one sub-problem is generated after the reduction:
  - binary search
  - root-finding algorithms (bisection — discussed in P6, or more efficient Newton's method)

  a recursive implementation can often be tail-call optimised.
- When the reduction results in two (or more, like in Karatsiba's number multiplication) smaller sub-problems:
  - Sorting algorithms, like Quick Sort and Merge Sort
  - Fast-Fourier Algorithm (first discovered by Carl Gauss)
  - Karatsuba's fast number multiplication algorithm (multiplication of two n-digit numbers can be done with $n^{\log_2 3} \approx n^{1.585}$ instead of n^2 single-digit multiplications)

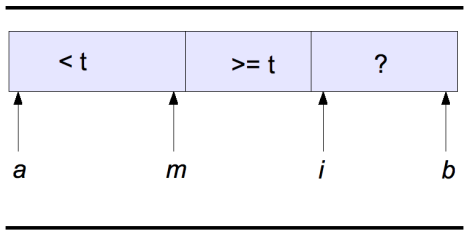
  one sometimes talks of **decrease and conquer** strategy.

Sometimes (rarely), the reduction is linear, like $n \to n-1$ in the Tower of Hanoi problem; usually it is geometrical (like in a geometrical progression), by the factor of $p > 1$ ($p = 2$ in Merge Sort, or Binary Search). In the latter case, the efficiency is $\mathcal{O}(N \cdot \log_p N)$, where the cost of solving the ultimately reduced problem (often having the size of 1) is $\mathcal{O}(1)$.

Recursive Algorithms: QuickSort

Recursion is a very elegant way of thinking about some algorithms and some recursive algorithms are very fast. However it is possible to write very slow recursive algorithms as well and some care is needed. (For example, you need to be aware that there is an overhead in making all of the method calls in the recursive factorial method.)

The most famous of all (and first recursive algorithm) is *QuickSort*, invented by Tony Hoare in 1962 before the very concept of recursions became known to computer professionals.



- The essence of *QuickSort* algorithm is the *divide-and-conquer* principle:
  1. select a pivot — index of an element whose value is between min and max of the array
  2. partition the array around the pivot — smaller elements go on the left, greater — to the right
  3. apply the *QuickSort recursively* to the left and the right subarrays
  4. recursion stops if `subarray.length` < 3

During the algorithm execution (after every partitioning is done) the following *invariant* must be maintained for an array `x[a..b]` with the pivot value `t = x[m]`):

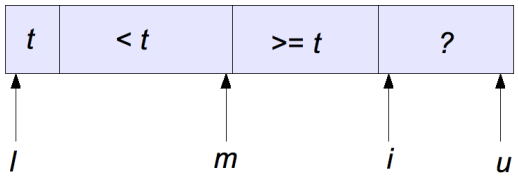
$\forall$ `i, a <= i < m, x[i] <= x[m]` and $\forall$ `i, m <= i <= b, x[m] <= x[i]`

QuickSort: the Pseudo-code

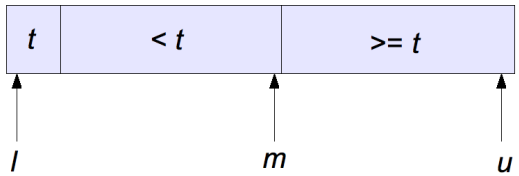
```
m = l; t = x[m]; // selecting pivot
for i = [l+1, u] //closed interval
   if x[i] < t      // before swapping, increment m to give the
      swap(++m, i) // left subarray a room for the new element
```

That's how the "detailed" partitioning algorithm looks like pictorially (code is QuickSort.java):

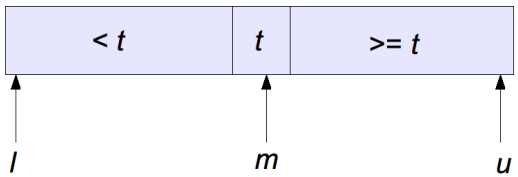
1. during the loop execution



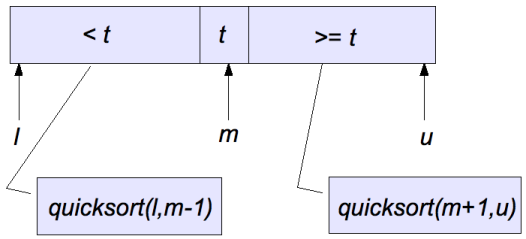
2. after the loop terminates



3. placing the pivot value in the right place

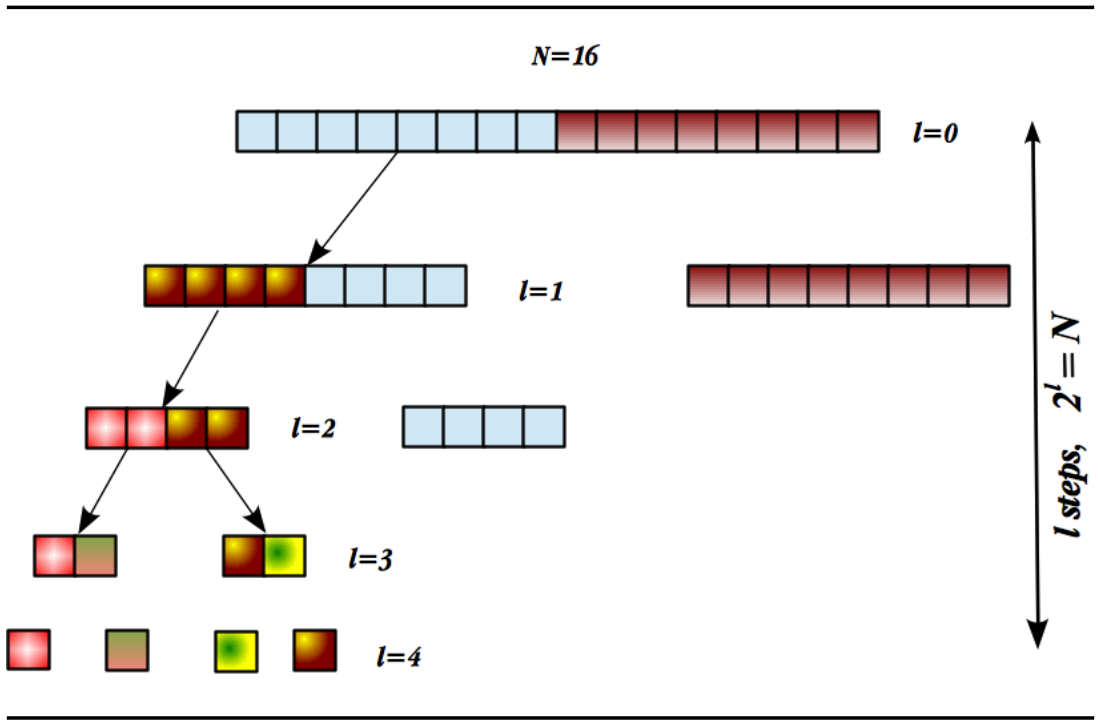


4. calling *QuickSort* on sub-arrays



How Quicksort Shines

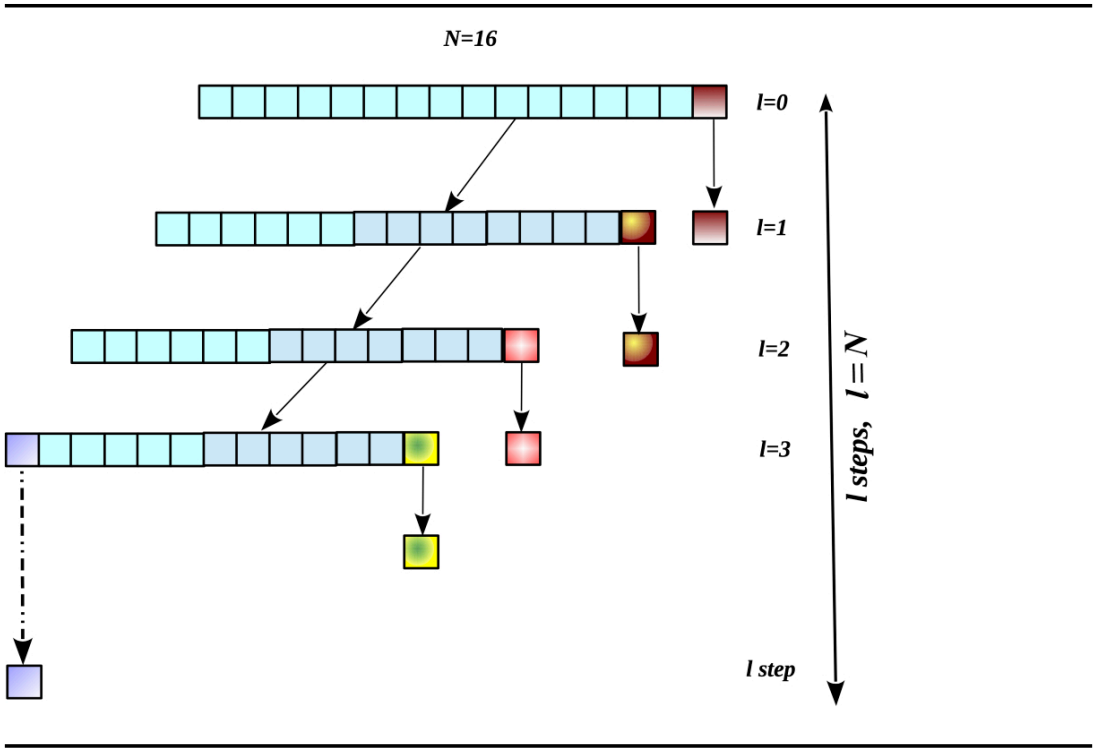
QuckSort is fast for generic (random-like) inputs, its efficiency is $\mathcal{O}(N \cdot \log_2 N)$.



- Total number steps in the level ℓ partitioning loops is the same, N
- For generic inputs, the subarrays have roughly the same length, $N/2^{\ell}$
- The complete recursion calls tree — all branches have roughly the same height, ℓ
- The sum of *all* partitioning steps at *all* levels is $N \cdot \ell = N \cdot \log_2 N$ ("swoosh!")

How *Quicksort* Flounders

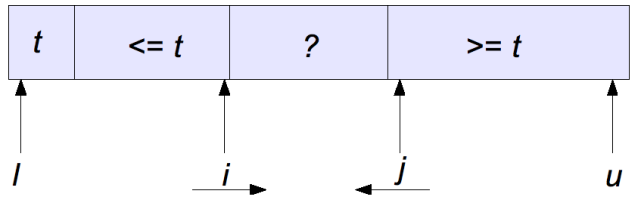
Yet, when an input becomes (almost) sorted, it degenerates to $\mathcal{O}(N^2)$. Why?



- For (almost) sorted arrays, partitioning results into one subarrays having only one element (may be a few)
- The recursion goes on, chipping off one element at a time
- The number of steps at every level is still N, but the number of levels is also N
- Hence the "snail"-like result: $N \cdot \ell = N \cdot N = N^2$ ("disaster!")

Fine-tuning QuickSort

The full analysis of a complex algorithm such as *QuickSort* is quite involved. Its weakness is taken care of to prevent performance problems, yet an industrial strength implementation (found in good libraries) is more than 200 LOC long (instead of puny 20–30 in an "educational" case like ours.)



Two-way partitioning in quicksort

- select the pivot *not* from the first element of the input, but from the middle, or randomly — this will improve sorting of nearly sorted inputs
- do partitioning with *two* inner loops going from the opposite ends towards each other — this will improve the performance for "all-equal" inputs

Robust QuickSort

There is a relatively simple way to avoid the drastic performance worsening of *QuickSort* for non-random input when the pivot selection is "attuned" to the value distribution in a way which results in partitioning always resulting into subarrays of very unequal length.

- The main weakness of *QuickSort* is sorting a large number of short arrays at the end of recursion chain.
- *InsertionSort*, on the other hand, is very effective if the input is an "array of subarrays" with each subarray unsorted, but all values in it are greater than the largest value in the preceding subarray, and not greater than the smallest value in the following subarray.
- Such "array of subarrays" is what `quicksort()` produces if stopped "prematurely", when `subarray.length >= cutoff > 2`.
- Completing the sorting with *InsertionSort* produces a *robust* routine:

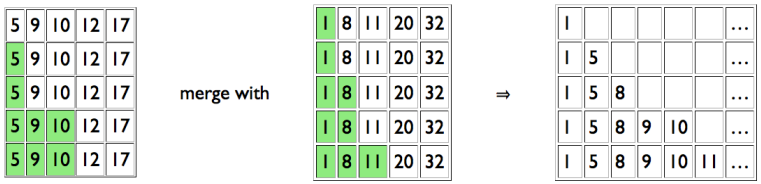
```
QuickSort(data, 0, data.length - 1, cutoff);
InsertionSort(data);
```

The hybrid *QuickSort* was used in `java.util.Array.sort()` methods (until JDK 7).

MergeSort

The elegant merging algorithm begins with... well, *merging two sorted arrays* which preserves the sorted order (the resulting "big" array is also sorted). The merging starts with choosing either the first element of one or the first element of the other array. Then one looks at the next two possible elements and chose one of them. The coding for this is straightforward but a little bit long-winded. From time to time you will be selecting several elements from one of the arrays before you select any from the other (for an implementation see MergeSortTest.java).

```
public static void mergeSort(
       int[] a, int from , int to) {
   if (from == to) return;
   int mid = (from + to)/2;
   mergeSort(a, from, mid);
   mergeSort(a, mid +1, to);
   merge(a, from, mid, to);
}
```



The idea of the merge sort is that you divide the original array into two parts, sort them and then merge them. If you use recursion, then you can effectively keep dividing until the initial arrays have size 1 so that they are trivially sorted and all you need to do is merge them. Because it uses the same principle "divide-and-conquer", the expected performance is the same as for *QuickSort* $\mathcal{O}(N \cdot \log_2 N)$. A hybrid of *MergeSort* and *InsertionSort* — so called *TimSort* — is used in `Array.sort()` since JDK 7.

TimSort

Until Java SDK 7, the `Arrays.sort`, `Collections.sort` and `List.sort` methods were using the hybrid QuickSort algorithm roughly described above. The modern versions of the API (at least, for *Oracle* and `openjdk`) use a different sorting algorithms, *Tim Sort*, which was developed not too long time ago. It combines MergeSort with InsertionSort "Tim" is a (modest) bloke who developed it.

The combination of *MergeSort* and *InsertionSort* is done as follows:

1. Segment (starting with 2) the original array into parts (of possibly equal length), store the segments starting positions and lengths on the stack
2. Apply the segmentation until the segment length drops down to chosen parameter value, for shorter segments use *InsertionSort* to sort them in-place
3. When two consecutive segments are sorted, merge them to preserve ordering using the values stored on the stack.

The key reason for *MergeSort* and *TimSort* to avoid deterioration of performance is the "always divide in two" principle (no pivot selection which *can* result into long branches of recursions).

TimSort was found to be buggy in SDK 7 implementation (some researches claim that fixing it in JDK 8, at least, in OpenJDK, did not solve all the problems).

Introsort

Introsort (also called *introspective sort*) is another hybrid sorting algorithm with the "usual" fast generic performance $\mathcal{O}(N \cdot \log N)$, and optimal degenerate case one. It uses QuickSort at the beginning, and when (if) the recursion depth exceeds $\log N$ (which indicates that QuickSort is about to go "awol") switches to *HeapSort* (the $\mathcal{O}(N \cdot \log N)$-algorithm which does not use divide-and-conquer recursive trick; it's briefly described in A9). Here is Introsort's pseudo-code:

```
procedure sort(A : array):
    let maxdepth = ⌊log(length(A))⌋ * 2
    introsort(A, maxdepth)


procedure introsort(A, maxdepth):
    n ← length(A)
    if n <= 1:
        return  // base case
    else if maxdepth = 0:
        heapsort(A)
    else:                       // assume that partition does pivot selection,
        p ← partition(A)        // and p is the final position of the pivot
        introsort(A[0:p], maxdepth - 1)
        introsort(A[p+1:n], maxdepth - 1)
```

The Introsort is the base for sort algorithm used in the venerable *Standard Template Library*, the collection of generic data structures and algorithm used in C++.

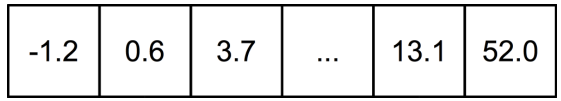
BinarySearch

This is another quintessential computational task — searching in a sorted database, the simplest case of which is a one-dimensional array. The algorithm seems almost trivial:

1. find the middle index `mid`, compare value of its element with the target
2. if the target is smaller than `x[mid]`, apply the *BinarySearch* to the left subarray
3. if the `target == x[mid]`, **Bingo!**
4. if the target is greater than `x[mid]`, apply the *BinarySearch* to the right subarray

---

- find position of target `a = 3.7` in the array $\longrightarrow$
- if found return the value of target index: `(2)`
- if not found return the special value `-1`



Despite such almost trivial principle, it's notoriously hard to achieve a correct implementation. D. Knuth, in the *Vol. 3*, reports that while the first *BinarySearch* algorithm was published in 1946, the first *correct* implementation was only achieved in 1962. The critical concept is a *loop invariant* —— see next slide and Ch.4 of "Programming Pearls" by Jon Bentley. *BinarySearch* is a $\mathcal{O}(\log_2 N)$ algorithm. It's often used "inside" other algorithms, *eg*, for finding roots of nonlinear equations via the Newton's method.

Rôle of invariants (*)

Implementing an algorithm in (pseudo-)code is often a more complex task than it may seem, even for simplest cases, like a *binary search algorithm*. Programs are like mathematical formulae. The latter can be (very) complex. Their validity is ensured by application of mathematical axioms and rules of logic *at every step* of formula manipulation.

The algorithm correctness is ensured in a similar way with the addition of an *invariant* which is a formal expression of the algorithm purpose.

Algorithm's invariant is a formal assertion about the program state which must be true at every stage of program execution. The invariant assertion involves input, program variables and output. A (single-threaded) program usually contains:

- Sequence control statements
- Selection control statements (conditionals)
- Iteration control statements (loops)
- Function calls

A program A built from these elements, $A = A_1 A_2 \dots A_i \dots A_n$, will be correct if each of them preserves the invariant, **and** the program itself terminates. (Termination of loops and function calls should be proved independently.)

- Correctness of **pre-condition** (which should include the invariant) $\{P_i\}$ for A_i implies
- Correctness of **post-condition** $\{Q_i\}$ (also includes the invariant), which serves as a precondition for $\{P_{i+1}\}$

Invariants in *BinarySearch* (*)

Correctness of a program (including an algorithm) is established by demonstrating that the full computational chain $A = A_1 A_2 \ldots A_i \ldots A_n$ (provided it terminates) preserves the invariant. Deduction $\{P_i\}A_i\{Q_i\}$ is performed using logic and "obvious" mathematics (Hoare's logic and Dijkstra's Weakest Precondition).

Choice of the invariant for a particular algorithm is more like an art than science. In the case of binary search in a linear array-like DS, the invariant can be formulated as follows (as taken from Jon Bentley's book "Programming Pearls"):

- The expression `mustbe(range)` is a short form of "if t is inside the array then it must be in *range*" (this is an implication, the preposition "if t is inside" matters!).
- The expression `cantbe(range)` is a short for "t cannot be in *range*".
- Inside a `for`-loop `{mustbe(range)}` serves as the *loop invariant*, which an *assertion* about the program which is true at the start and at the end every loop iteration.

If the algorithm preserves the invariant (and if it terminates), we can be ("almost") assured that the result is correct. The algorithm analysis ("walk-through") must establish that the invariant is never broken (including the proof that the algorithm halts at some point) at each execution step.

Breaking by blindness to obvious (*)

Again

- `mustbe(range)` means that the search target `t` must be inside the `range l...u`
- `cantbe(range)` means that `t` cannot be in `range`

Here is the *BinarySearch* pseudo-code with the invariant:

```
01 { mustbe(0, n-1) }
02 l = 0; u = n-1;
03 { mustbe(l, u) }
04 loop
05   { mustbe(l, u) }
06   if l > u
07     { l > u && mustbe(l, u) }
08     { t is not in the array }
09     p = -1; break
10   { mustbe(l, u) && l <= u }
11   m = (l + u) / 2
12   { mustbe(l, u) &&
                  l <= m <= u }
13   case
14     x[m] < t:
15       { mustbe(l, u) && cantbe(0, m) }
16       { mustbe(m+1, u) }
17       l = m+1
18       { mustbe(l, u) }
19     x[m] == t:
20       { x[m] == t }
21       p = m; break
22     x[m] > t:
23       { mustbe(l, u) && cantbe(m, n) }
24       { mustbe(l, m-1) }
25       u = m-1
26       { mustbe(l, u) }
27   { mustbe(l, u) }
```

Latest Bug in *BinarySearch*

Bugs can stay hidden for a long time until execution environment changes and they get "activated". This happened to Java's implementation of the binary search as it was used in *java.util.Arrays* `binarySearch()` methods:

```
1:      public static int binarySearch(int[] a, int key) {
2:          int low = 0;  int high = a.length - 1;
3:          while (low <= high) {
4:              int mid = (low + high) / 2; // Where the problem lurks!
5:              int midVal = a[mid];
6:              if (midVal < key)
7:                  low = mid + 1;
8:              else if (midVal > key)
9:                  high = mid - 1;
10:             else
11:                 return mid; // key found
12:         }
13:         return -(low + 1);  // key not found.
14:     }
```

The bug struck in 2006, when the size of the search array became too large and some `int` values exceeded the allowed range. The solution is to replace the **line 4** on:

```
int mid = low + ((high - low) / 2);    OR    int mid = (low + high) >>> 1;
```

Where to look for this topic in the textbook?

Very little about recursive algorithms and algorithms in general is discussed in the textbook (it teaches *users* how to program, not *developers* of programming resources, like libraries and API). You should seek other sources of knowledge in these areas, both in printed form and (much more and growing) online.

- Hortsmann's Core Java for the Impatient, Ch. 3.3, 3.8
- Oracle's Java Tutorial chapter Algorithms — again, it deals with the JFC usage, not much with the intricacies of generic algorithm implementation.
- One modern (and relatively light) text is "Algorithms in a Nutshell", 2ed, by George T. Heineman, Gary Pollice, and Stanley Selkow (O'Reilly, 2017)