COMP26120: Algorithms and Imperative Programming

Basic sorting algorithms

Ian Pratt-Hartmann

Room KB2.38: email: ipratt@cs.man.ac.uk

2017–18
• Reading for this lecture (Goodrich and Tamassia):
  • Secs. 8.1, 8.2, 8.3 (pp. 241–258).
  • Sec. 1.1.5 (pp. 11–16).
Outline

Quicksort

Mergesort

A lower bound?
• Consider the problem of sorting a list of numbers (in ascending order).
• Quicksort is a sorting algorithm which works well in practice.
  
  quicksort(L)
  if length of \( L \leq 1 \)
      return \( L \)
  remove the first element, \( x \), from \( L \)
  \( L_\leq := \) elements of \( L \) less than or equal to \( x \)
  \( L_\geq := \) elements of \( L \) greater than \( x \)
  \( L_\ell := \) quicksort(\( L_\leq \))
  \( L_r := \) quicksort(\( L_\geq \))
  return \( L_\ell + [x] + L_r \)
  
• The element \( x \) is sometimes referred to as the pivot.
Example:

\begin{verbatim}
quicksort([x|L])
    if length of L ≤ 1
        return L
    remove x from L
    compute L_≤, L_>
    L_≤ := quicksort(L_≤)
    L_≥ := quicksort(L_≥)
    return L_≤ + [x] + L_≥
end
\end{verbatim}

quicksort([2,9,1,3,4,0])
quicksort([1,0])
quicksort([0])
quicksort([ ])
quicksort([9,3,4])
quicksort([3,4])
quicksort([ ])
quicksort([4])
quicksort([ ])

- **Example:**

  ```java
quicksort([x|L])
  if length of L ≤ 1
    return L
  remove x from L
  compute Lₗ, Lᵣ
  Lₗ := quicksort(Lₗ)
  Lᵣ := quicksort(Lᵣ)
  return Lₗ + [x] + Lᵣ
  end
```

- `quicksort([2,9,1,3,4,0])`
- `quicksort([1,0])`
- `quicksort([0]) [0]`
- `quicksort([9,3,4])`
- `quicksort([3,4])`
- `quicksort([ ])`
• Example:

```plaintext
quicksort([x|L])
  if length of L ≤ 1
    return L
  remove x from L
  compute L_≤, L>
  L_ℓ := quicksort(L_≤)
  L_r := quicksort(L_>)
  return L_ℓ + [x] + L_r
end
```

quicksort([2,9,1,3,4,0])
quicksort([1,0])
quicksort([0])
quicksort([9,3,4])
quicksort([3,4])
quicksort([0])
quicksort([0])
quicksort([4])
quicksort([4])
Example:

\[
\text{quicksort}([x|L])
\]

\[
\begin{align*}
\text{if length of } L &\leq 1 \\
\text{return } L
\end{align*}
\]

\[
\begin{align*}
\text{remove } x &\text{ from } L \\
\text{compute } L_{\leq}, L_{>}
\end{align*}
\]

\[
\begin{align*}
L_{\ell} &:= \text{quicksort}(L_{\leq}) \\
L_r &:= \text{quicksort}(L_{>})
\end{align*}
\]

\[
\text{return } L_{\ell} + [x] + L_r
\]

end

\[
\text{quicksort}([2,9,1,3,4,0])
\]

\[
\begin{align*}
\text{quicksort}([1,0]) &\to [0,1] \\
\text{quicksort}([0]) &\to [0] \\
\text{quicksort}([ ]) &\to [ ] \\
\text{quicksort}([9, 3, 4]) &\to [9, 3, 4] \\
\text{quicksort}([3, 4]) &\to [3, 4] \\
\text{quicksort}([ ]) &\to [ ]
\end{align*}
\]
• Example:

```plaintext
quicksort([x|L])
    if length of L ≤ 1
        return L
    remove x from L
    compute L≤, L>
    Lℓ := quicksort(L≤)
    Lr := quicksort(L>)
    return Lℓ + [x] + Lr
end
```

```plaintext
quicksort([2,9,1,3,4,0])
quicksort([1,0]) [0,1]
quicksort([0]) [0]
quicksort([ ]) []
quicksort([9, 3, 4])
quicksort([3, 4])
quicksort([ ]) []
quicksort([4])
quicksort([ ]) 
```
Example:

```plaintext
quicksort([x|L])
    if length of L ≤ 1
        return L
    remove x from L
    compute L_≤, L_{>}
    L_≤ := quicksort(L_≤)
    L_{>} := quicksort(L_{>})
    return L_≤ + [x] + L_{>}
end
```

```
quicksort([2,9,1,3,4,0])
quicksort([1,0]) [0,1]
quicksort([0]) [0]
quicksort([ ]) []
quicksort([9, 3, 4])
quicksort([3, 4])
quicksort([ ]) []
quicksort([4]) [4]
quicksort([ ]) []
```
**Example:**

\[
\text{quicksort}([x|L])
\begin{align*}
\text{if } \text{length of } L &\leq 1 \\
\text{return } L &\quad \text{quicksort}([1,0]) \quad [0,1] \\
\text{remove } x \text{ from } L &\quad \text{quicksort}([0]) \quad [0] \\
\text{compute } L_\leq, L_> &\quad \text{quicksort}([ ]) \quad [ ] \\
L_\ell := \text{quicksort}(L_\leq) &\quad \text{quicksort}([9, 3, 4]) \\
L_r := \text{quicksort}(L_>) &\quad \text{quicksort}([3, 4]) \quad [3, 4] \\
\text{return } L_\ell + [x] + L_r &\quad \text{quicksort}([ ]) \quad [ ] \\
\end{align*}
\text{end}
\]
Example:

```plaintext
quicksort([x|L])
    if length of L ≤ 1
        return L
    remove x from L
    compute L_≤, L_>
    L_ℓ := quicksort(L_≤)
    L_r := quicksort(L_>)
    return L_ℓ + [x] + L_r
end
```

```plaintext
quicksort([2,9,1,3,4,0])
quicksort([1,0]) [0,1]
quicksort([0]) [0]
quicksort([ ]) []
quicksort([9, 3, 4])
quicksort([3, 4]) [3, 4]
quicksort([ ]) []
quicksort([4]) [4]
quicksort([ ]) []
```
Example:

```plaintext
quicksort([x|L])
    if length of L ≤ 1
        return L
    remove x from L
    compute L_≤, L_>
    L_ℓ := quicksort(L_≤)
    L_r := quicksort(L_>)
    return L_ℓ + [x] + L_r
end
```

```plaintext
quicksort([2,9,1,3,4,0])
quicksort([1,0]) [0,1]quicksort([0]) [0]
quicksort([ ]) []
quicksort([9, 3, 4]) [3, 4, 9]quicksort([3, 4]) [3, 4]quicksort([ ]) []
quicksort([4]) [4]quicksort([ ]) []
quicksort([ ]) []
```
Example:

```plaintext
quicksort([x | L])
    if length of L ≤ 1
        return L
    remove x from L
    compute L_≤, L_> 
    L_ℓ := quicksort(L_≤)
    L_r := quicksort(L_>)
    return L_ℓ + [x] + L_r
```
Let’s see how much work is done:

The worst case occurs when, for each recursive call, one of \( L_{\leq} \) or \( L_{>\} \) is empty.

Here \( n \) recursive calls are made (ignoring calls with \([\;]\)) , with the argument one element shorter each time.

Before each recursive call, \( L_{\leq} \) and \( L_{>\} \) must be calculated, requiring \( O(|L|) \) steps.

So if \(|L| = n\), total work is order

\[
n + n - 1 + \cdots + 1 = \frac{1}{2}n(n + 1)
\]

i.e. \( O(n^2) \) (because \( O\left(\frac{1}{2}n(n + 1)\right) = O(n^2)\)).
Outline

Quicksort

Mergesort

A lower bound?
• Here is an algorithm with lower complexity.
• First, consider the problem of merging two sorted list to form a third sorted list.

\[ \text{merge}([1, 3, 5], [0, 2, 4, 6, 7]) \Rightarrow [0, 1, 2, 3, 4, 5, 6, 7] \]

• This algorithm will work.

\[
\begin{align*}
\text{merge}(L_1, L_2) \\
\text{if } L_1 = [] & \quad \text{return } L_2 \\
\text{if } L_2 = [] & \quad \text{return } L_1 \\
x_i = \text{first element of } L_i & \quad (i = 1, 2) \\
L'_i = L_i \text{ minus first element} & \quad (i = 1, 2) \\
\text{if } x_1 \leq x_2 & \\
\quad \text{return } [x_1] + \text{merge}(L'_1, L_2) \\
\quad \text{return } [x_2] + \text{merge}(L_1, L'_2) \\
\end{align*}
\]
• When `merge(L_1, L_2)` is called, at most one recursive call is made, in which `|L_1| + |L_2|` decreases by 1.
• Therefore, at most \(O(n)\) recursive calls are made, where \(n = |L_1| + |L_2|\) is the length of the input.
• A constant number of operations is executed for each recursive call.
• Therefore, at it takes most \(O(n)\) time to run.
• We can now present our sorting algorithm

\[
\text{mergeSort}(L) \\
\quad \text{if } |L| \leq 1 \\
\quad \quad \text{return } L \\
\text{Split } L \text{ into two roughly equal halves } L = L_\ell + L_r \\
\text{return } \text{merge(mergeSort}(L_\ell),\text{mergeSort}(L_r))
\]

\end{aligned}

• This algorithm clearly returns a sorted list with exactly the original elements.
• How many times is the algorithm called recursively?
• The following analysis gives a rather disappointing bound:
  • Each recursive call gives rise to two others at one greater depth of recursion.
  • Thus, each depth of iteration, there are twice as many recursive calls.
  • The maximum depth of recursion is $\lceil \log_2 n \rceil$.
  • Therefore, the number of calls is $2^{\lceil \log_2 n \rceil} \leq 2n$.
  • (Actually, a better bound is $n - 1$: can you see why?)
• The time taken to merge is at most $O(n)$, so this suggests (prima facie) a complexity bound of $O(n^2)$. 
• But in fact it’s not that bad.

The total lengths of lists processed at each level of recursion is constant at $|L| = n$.

And the total amount of work done for each call is linear in the lengths of the arguments.

The number of times $L$ can be halved is $O(\log n)$.

Hence, the time complexity of $\text{mergeSort}$ is $O(n \log n)$. 
• Or do some algebra. Let the time taken by mergeSort on any list of length $n$ be bounded by (worst case), $t(n)$. Then, ignoring constant factors

$$t(n) = 2t\left(\frac{n}{2}\right) + n$$

and, without loss of generality, we may as well assume that $t(2) \leq 2$.

• A simple induction shows that

$$t(n) \leq n \log_2 n.$$

For, $n > 2$ (and cheating quite a lot), we have

$$t(n) = 2t\left(\frac{n}{2}\right) + n$$

$$\leq 2 \frac{n}{2} \log_2 \left(\frac{n}{2}\right) + n \quad \text{(ind. hyp.)}$$

$$= n \log_2 n.$$
Outline

Quicksort

Mergesort

A lower bound?
• Can we do any better than $O(n \log_2 n)$?

• In the study of algorithms, lower complexity bounds are in general extraordinarily hard to obtain.

• In the case of sorting, however, we have a qualified lower bound:

  *Any algorithm which sorts a list using only number-comparison operations requires time at least $n \log_2 n$ to run.*

• Let us see why this is so.
• First some basic facts about trees.
• Suppose we have a full binary tree of depth $d$ with $n$ vertices in total, of which $\ell$ are leaves.

In this example, $d = 3$, $n = 15$ and $\ell = 8$.
• Starting with the root at level 0, the number of vertices on each level $k$ is $2^k$. Hence

$$\ell = 2^d$$

$$n = \sum_{k=0}^{d} 2^k = 2^{d+1} - 1.$$  

• Otherwise expressed:

$$d = \log_2 \ell$$

$$d = \log_2(n + 1) - 1.$$
• If the tree is binary branching, but not full, then these equalities are replaced by inequalities.

\[ d = 3 \]

Here we are missing some vertices. Hence:

\[ \ell \leq 2^d \quad n = \leq 2^{d+1} - 1. \]

• Otherwise expressed:

\[ d \geq \log_2 \ell \quad d \geq \log_2(n + 1) - 1. \]
• Now suppose we have an algorithm which sorts a list by making comparisons and branching as a result of that comparison.

• The possible runs of that algorithm may be arranged as a binary tree.

• Assume without loss of generality, that the input of length $n$ are the integers $1–n$ in some order $\pi(1), \ldots, \pi(n)$, where $\pi$ is a permutation.

• The algorithm will then apply the inverse permutation $\pi^{-1}$ to sort the list.
• There are $n!$ permutations of the numbers 1–$n$, each requiring a different output, and hence $n!$ leaves in the computation tree.

$$t(n) = d$$

• The maximum running time, $t(n)$, on inputs of length $n$ is the (maximum) depth of the tree.

• From our inequality $d \geq \log_2(\ell)$ we obtain, assuming $n$ even:

$$t(n) \geq \log_2(n!) \geq \log_2 \left( \left( \frac{n}{2} \right)^{\frac{n}{2}} \right)$$

$$= \frac{n}{2} \log_2 \left( \frac{n}{2} \right) = \frac{1}{2} n \log_2(n) - 1.$$
• The following is a very handy way of talking about lower bounds.

• If \( f : \mathbb{N} \to \mathbb{N} \) is a function, then \( \Omega(f) \) denotes the set of functions:

\[
\{ g : \mathbb{N} \to \mathbb{N} \mid \exists n_0 \in \mathbb{N} \text{ and } c \in \mathbb{R}^+ \text{ s.t. } \forall n > n_0, \ g(n) \geq c \cdot f(n) \}.
\]

• Thus, \( \Omega(f) \) denotes a set of functions, intuitively, the functions that grow essentially at least as fast as \( f \).
• Notice that for $n \geq 4$, $\frac{1}{2} n \log_2(n) \geq n$.

• In particular, for sufficiently large $n$,

$$\frac{1}{2} n (\log_2(n) - 1) \geq \frac{1}{4} n \log_2(n).$$

• That is,

$$\frac{1}{2} n (\log_2(n) - 1) \in \Omega(n \log_2(n))$$

• Thus, we are guaranteed that any sorting algorithm based on comparisons has running time (in) $\Omega(n \log_2(n))$.

• Warning, this doesn’t provide a guarantee of the complexity of any algorithm whatsoever. On the other hand, no one has done any better so far . . .