Proof that $O(g(n)) = O'(g(n))$

Given $g: \mathbb{N} \to \mathbb{Z}^+$, we will first show $O'(g(n)) \subseteq O(g(n))$; then we'll show that $O(g(n)) \subseteq O'(g(n))$.

Let $f \in O'(g(n))$ where $f: \mathbb{N} \to \mathbb{Z}^+$.

By the definition of $O'$, $\exists c > 0 \ \forall n, 0 < f(n) \leq c \ g(n)$.
Let $n_0 = 0$. Clearly $\forall n > n_0, 0 \leq f(n) \leq c \ g(n)$. Thus, $f \in O(g(n))$.

Now let $f \in O(g(n))$ where $f: \mathbb{N} \to \mathbb{Z}^+$.

By definition, $\exists c > 0 \ \exists n_0 \ \forall n > n_0, 0 \leq f(n) \leq c \ g(n)$. (Equation I)
Let $c' = \max(c, f(0), f(1), \ldots, f(n_0))$. Clearly $c' > 0$.

Given any $n$, if $n \leq n_0$, then $f(n) \leq c'$ (by the def. of max) and so $0 < f(n) \leq c' \ g(n)$ (since $f(n)$ and $g(n)$ are both $\geq 1$.)

Conversely, if $n > n_0$, then $f(n) \leq c \ g(n)$ from Equation I.

But $0 < f(n)$ and and $c' \geq c$, so $0 < f(n) \leq c' \ g(n)$.

Thus, we've shown $\forall n, 0 < f(n) \leq c' \ g(n)$, and so $f \in O'(g(n))$.

Q.E.D
Overview

• Quicksort and Heapsort are comparable
  - Both use only binary comparisons
  - Both sort in-place

• Heapsort:
  - worst-case complexity is \( \Theta(n \log n) \)

• Quicksort:
  - worst-case complexity is \( \Theta(n^2) \).
  - average-case complexity is \( \Theta(n \log n) \)
  - probabilistic analysis is also \( \Theta(n \log n) \)

• Yet Quicksort is often considered superior.

Priority queue

A “task” is an object with a “key” field

Key of task \( x \) is \( x.key \) or \( key(x) \) (or whatever style you like).

We’ll just worry about the priorities.

A (max-) priority queue is a data structure that has the following operations:

- Insert\((S, x)\) - add task \( x \) to the queue \( S \).
- Extract-Max\((S)\) - return the task with largest key and remove it from heap.
- Max\((S)\) - return task with largest key (don’t remove it).
- Increase-Key\((S,x,k)\) - Increase \( x \)’s key to be \( k \)
  (return error code if \( x.key \) is already > \( k \).)
Heaps can implement priority queues

A **heap** is a binary tree:
- All levels except bottom are completely filled in.
- Bottom level is filled in from left (no holes).
- Has heap property: parent’s key ≥ either child’s

A heap can be stored in an array H:
- Root is H[1].
- Left child of H[k] is H[2k]
- Right child of H[k] is H[2k+1]

Insert(S, x) - add task x to the queue S.
- Add x as new last node.
- “Bubble up” to re-establish heap property.

Extract-Max(S) - return the task with largest key and remove it from heap.
- Pull task from top of heap (it has largest key).
- Replace it with the last node of heap.
- “Bubble down” (heapify) to re-establish heap property.

Increase-Key(S, x, k) - Increase x’s key to be k.
- Report error if x.key > k
- Set x.key = k.
- “Bubble up” to re-establish heap property.
Exercise

- Pick a random permutation of \{1,2,3,4,5,6\}
- Insert these priorities into heap in the chosen order.
- Now do two Extract-Max’s.

Heapsort

- Insert all the \( n \) data items into a heap, then extract them all.
- Insert and Extract-Max operations use at most \( c \lg n \) time.

Aside: does this follow from our theorem, “If \( T \) is a non-empty binary tree of height \( h \), then \( T \) has fewer than \( 2^{h+1} \) nodes”? ??

- So \( T(n) = \text{time to sort } n \text{ items} < 2^n c \log n \),
  \( T(n) \in O(n \log n) \).
**Build-Heap**

Builds heap from set $S$ using $O(n)$ operations ($n=|S|$).

Still, Heapsort's asymptotic complexity is $O(n \lg n)$, since you need $n$ Extract-Max's.

Stuffs $S$ into a binary tree, then massages it *from last parent to first* to establish heap property.

No comparisons needed for leaves.
Each node at level $h-i$ needs at most $2i$ comparisons.

Comparisons bounded by:

$$n/2 \times 2^1 + n/4 \times 2^2 + n/8 \times 2^3 + \ldots + 1 \times 2^\lg n$$

$$= 2n \left(1/2 + 2/4 + 3/8 + 4/16 + \ldots + \lg n/n\right)$$

---

**Summing $i/2^i$**

$$1/2 + 1/4 + 1/8 + 1/16 + 1/32 + \ldots \leq 1$$

$$1/4 + 1/8 + 1/16 + 1/32 + \ldots \leq 1/2$$

$$1/8 + 1/16 + 1/32 + \ldots \leq 1/4$$

$$1/16 + 1/32 + \ldots \leq 1/8$$

$$\vdots$$

$$\vdots$$

$$1/2 + 2/4 + 3/8 + 4/16 + 5/32 + \ldots \leq 2$$
The two ways to build a heap

• Use Build-Heap
  \[ T(n) < 2n \left( \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \ldots \right) = 2n \times 2 = 4n \]

• Make repeated calls on Insert(H,x)
  In the worst case, each insertion requires “bubbling up” all the way to root.
  For half the nodes, this takes \( -2 \) comparisons.
  So \( T(n) > \frac{n}{2} (\log n - 2) \), i.e. \( T(n) \in \Omega(n \log n) \)
  (Average case may not be so bad.)

• Intuition why Build-Heap is (or might be) better:
  Most nodes in a heap are close to the leaves.
  Most nodes in a heap are far from the root.

Quicksort

Basic idea:
  Split w.r.t \( A[1] \)
  Arrange \( A \) as:
  \[
  \begin{array}{c}
  \text{small elements} \ (\leq A[1]) \\
  \text{big} \ (\geq A[1]) \\
  \end{array}
  \]

  Recursively arrange “small” part and “big” parts.

  Doesn’t need extra array.
  Keep pointers to current ends of small & big parts.

  “Pick up” \( A[1] \) (splitter) and \( A[n] \) (current element), leaves space in to deposit element in either part
drop current element appropriately; pick up next innermost.
Worst-case Quicksort complexity

What happens if A is already sorted?

\[ T(n) = T(1) + T(n-1) + (n-1) \]
\[ T(n) = (n-1) + (n-2) + ... = n(n-1)/2 \]

Similar problem if A is nearly sorted.

Is this unlikely?

Can a “hack” help?

E.g., splitter = median(first, middle, last)?

Average-case Quicksort complexity

• Intuitively, hope that on “random” input, most of the splits aren’t too uneven.
  - If, say, 50% of time, splits are no worse than 1/10 vs 9/10, you might be OK
    • As long as the bad splits are evenly spread around, this kind of looks like the recurrence:
      \[ T(n) < T(n/10) + T(9n/10) + 2n \]
    • Actually, on random input, things are more even.
    • But this is far from a proof!
**Digression: Probability**

A **sample space** $S$ is a set of “elementary events”.

An **event** is a subset of $S$.

A **probability distribution** is a function $Pr$ from events to real numbers in $[0,1]$ which satisfies certain properties.

If $S$ is **discrete** (finite or countably infinite), these properties amount to:
- If $s \subseteq S$, $Pr(s) = \sum Pr(e)$, where sum is over $e \in s$.
- $Pr(S) = 1$.

If $|S| = n$ and $Pr(e) = 1/n$ for all $e \in S$, $Pr$ is called **uniform**.

Note: We write $Pr(s)$ or $Pr(e)$ rather than $Pr(s)$ or $Pr({e})$.

---

**Probability factoids**

Probabilities are often abused

What does “There’s a 30% chance of rain” mean??

It is not possible to have a uniform probability space on $\mathbb{N}$ or $\mathbb{Z}$ (or any countably infinite set).

- “Pick $n \in \mathbb{N}$ with equal probability” is meaningless.

There are 3 reasons for using uniform probabilities:

1. You control selection of events and ensure uniformity.
2. Someone else assures uniformity (you can shift blame.)
3. You can’t think of anything better.

**Reason #3 is a lousy reason!!**
Random Variables

A (discrete) random variable $X$ is a function from elementary events in a sample space to $\mathbb{R}$.

Examples:
- $X(p) = p$'s height for $p \in S = \text{people in this class}$.
- $T_n(i) = \text{time to solve instance } i \text{ of size } n \text{ problem}$.

Notation: "$X>72$" is the event \{ $p \in S \mid X(p)>72$ \}.

$X+Y$ is function $(X+Y)(e) = X(e) + Y(e)$.

The expectation $E[X]$ of $X$ is $\sum_{e \in S} X(e) \Pr\{e\}$

$E[X]$ is the average, weights by the probabilities.


Average-case Quicksort complexity

Important insight:

Let $x$ & $y$ be two elements with $x \leq y$.

- If we pick some pivot $z$ s.t. $x \leq z \leq y$ before we pick either $x$ or $y$, then we'll never compare $x$ to $y$.
- Assume we pick the pivots randomly from all the candidates in an unpartitioned group.
- If $x$ and $y$ are $k$ places apart in the final sorted order, the chance of picking one of $x$ or $y$ before picking $z$ between them is $2/(k+1)$.
- So the further apart two elements are in the final order, the less likely they will be compared.
Average-case Quicksort complexity

“If x and y are k places apart in the final sorted order, there chance of picking one of x or y before picking something between 2/k+1.”

There are: 1 pair $n-1$ apart, $1 \times \frac{2}{n}$
2 pairs $n-2$ apart, $2 \times \frac{2}{(n-1)}$
...
$n-1$ pairs 1 apart, $(n-1) \times 2/2$

Expected number of comparisons is the total
(Uses fact: $E(\sum C_{xy}) = \sum E(C_{xy})$)

This total is $\Theta(n \log n)$

Randomized algorithm

- Quicksort has “bad” problem instances.
- A randomized algorithm makes random choices after the instance is selected.
  - E.g. it can choose splitter via coin-flipping.
  - Algorithm can ensure each possible choice has equal probability.
  - An adversary can’t find any particularly bad instance.
Probabilistic vs Average Analysis

Average-case analysis:
Let \( P_n = \{I_1, I_2, \ldots, I_k\} \) be instances of size \( n \)
\( P_n \) is the sample space.
Uniform probabilities assumed.
Is there a good reason for this?
Average performance \( T(n) \) is \( E(P_n) \).

Probabilistic analysis:
For each instance \( I \), let \( P_I = \{r_1, r_2, \ldots, r_k\} \) be possible choices made in randomized part of algorithm.
Assume uniform probabilities on \( P_I \). (Is there a good reason?)
Define \( Time(I) = E(P_I) \) (the expected time on instance \( I \)).
Probabilistic \( T(n) \) is \( \max\{Time(I)\} \) \( \max \) is over all instances of size \( n \).

Probabilistic analysis of randomized Quicksort

• For each instance of sorting, randomized Quicksort has expected time \( \Theta(n \lg n) \).
  - The same analysis as for that average time of (non-randomized) Quicksort.

• Warning: Result only holds for "truly" random choices of pivot elements.

  Amazing paper by Karloff & Raghavan shows:
  for any standard linear congruential pseudo-random number generator (e.g. Unix’s “rand”),
  there is a (carefully constructed) “bad” sorting instance
  that, averaged over all PRNG “seeds”
  has expected time \( O(n^2) \)
Stability

A sorting algorithm is **stable** if elements with equal keys stay in the same order.

If Sort({5, 8, 3, 10, 8}) returned {10, 8, 8, 5, 3}, it wouldn’t be stable (since 8 and 8 got swapped).

Stable sorting is useful; e.g. you might want to first sort by one key field, then by another.

- Is Heapsort stable?
- Is Quicksort stable?

Why use Quicksort?

- “Quicksort has tight code, so the hidden constant factor in its running time is small” (Text, pg 125).
  - Doesn’t heapsort also??

- “[Quicksort] works well even in virtual memory environments.” (Text, pg 145).
  - We’ll see what that means!
Glossary (in case symbols are weird)

\(\subseteq\subseteq\forall\exists\Theta\Omega\Sigma\geq\leq\approx\#\mathbb{N}\mathbb{R}\mathbb{Q}\mathbb{Z}\)

\(\subseteq\) subset  \(\in\) element of
\(\forall\) for all  \(\exists\) there exists
\(\Theta\) big theta  \(\Omega\) big omega  \(\Sigma\) summation
\(\geq\)  \(\leq\)  \(\approx\) about equal
\(\neq\) not equal  \(\mathbb{N}\) natural numbers(\(\mathbb{N}\))
\(\mathbb{R}\) reals(\(\mathbb{R}\))  \(\mathbb{Q}\) rationals(\(\mathbb{Q}\))  \(\mathbb{Z}\) integers(\(\mathbb{Z}\))