COMP 761: Lecture 22 – Sorting

David Rolnick

October 26, 2020

Problem

Prove that for integers $n \geq 2$, we have:

$$\frac{1}{2} + \cdots + \frac{1}{n} \leq \log n \leq 1 + \frac{1}{2} + \cdots + \frac{1}{n-1}.$$

(Please don't post your ideas in the chat just yet, we'll discuss the problem soon in class.)

Course Announcements

Course Announcements

- Office hours today right after class!

Asymptotic notation: $O(\cdot)$

Asymptotic notation: $O(\cdot)$

- We write $f(x) = O(g(x))$ (*big O*) if there is a constant $c > 0$ and a value x_0 such that $f(x) \le cg(x)$ whenever $x > x_0$.

Asymptotic notation: $O(\cdot)$

- We write $f(x) = O(g(x))$ (*big O*) if there is a constant $c > 0$ and a value x_0 such that $f(x) \leq cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = 2x^2 + 1$ and $g(x) = x^2$, we have

$$2x^2 + 1 = O(x^2),$$

because for $2x^2 + 1 < 3x^2$ for x sufficiently large.

Asymptotic notation: $O(\cdot)$

- We write $f(x) = O(g(x))$ (*big O*) if there is a constant $c > 0$ and a value x_0 such that $f(x) \leq cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = 2x^2 + 1$ and $g(x) = x^2$, we have

$$2x^2 + 1 = O(x^2),$$

  because for $2x^2 + 1 < 3x^2$ for x sufficiently large.
- Note that we also have:

$$2x^2 + 1 = O(x^3).$$

Asymptotic notation: $O(\cdot)$

- We write $f(x) = O(g(x))$ (*big O*) if there is a constant $c > 0$ and a value x_0 such that $f(x) \leq cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = 2x^2 + 1$ and $g(x) = x^2$, we have

$$2x^2 + 1 = O(x^2),$$

  because for $2x^2 + 1 < 3x^2$ for x sufficiently large.
- Note that we also have:

$$2x^2 + 1 = O(x^3).$$

- Some people write $f(x) \in O(g(x))$, but we will use $=$

Asymptotic notation: $O(\cdot)$

- We write $f(x) = O(g(x))$ (*big O*) if there is a constant $c > 0$ and a value x_0 such that $f(x) \leq cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = 2x^2 + 1$ and $g(x) = x^2$, we have

$$2x^2 + 1 = O(x^2),$$

  because for $2x^2 + 1 < 3x^2$ for x sufficiently large.
- Note that we also have:

$$2x^2 + 1 = O(x^3).$$

- Some people write $f(x) \in O(g(x))$, but we will use $=$
- What could we write for $x^3 + 100x^2$?

Asymptotic notation: $O(\cdot)$

- We write $f(x) = O(g(x))$ (*big O*) if there is a constant $c > 0$ and a value x_0 such that $f(x) \leq cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = 2x^2 + 1$ and $g(x) = x^2$, we have

$$2x^2 + 1 = O(x^2),$$

because for $2x^2 + 1 < 3x^2$ for x sufficiently large.
- Note that we also have:

$$2x^2 + 1 = O(x^3).$$

- Some people write $f(x) \in O(g(x))$, but we will use $=$
- What could we write for $x^3 + 100x^2$?
- We could say:

$$x^3 + 100x^2 = O(x^3), \text{ (or } O(x^4)\text{, etc.).}$$

Asymptotic notation: $O(\cdot)$

- We write $f(x) = O(g(x))$ (*big O*) if there is a constant $c > 0$ and a value x_0 such that $f(x) \leq cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = 2x^2 + 1$ and $g(x) = x^2$, we have

$$2x^2 + 1 = O(x^2),$$

  because for $2x^2 + 1 < 3x^2$ for x sufficiently large.
- Note that we also have:

$$2x^2 + 1 = O(x^3).$$

- Some people write $f(x) \in O(g(x))$, but we will use $=$
- What could we write for $x^3 + 100x^2$?
- We could say:

$$x^3 + 100x^2 = O(x^3), \text{ (or } O(x^4)\text{, etc.).}$$

- This is because for x big enough, $2x^3 \geq x^3 + 100x^2$.

Asymptotic notation: $o(\cdot)$

Asymptotic notation: $o(\cdot)$

- We write $f(x) = o(g(x))$ (*little o*) if for every constant $c > 0$ there is a value x_0 such that $f(x) < cg(x)$ whenever $x > x_0$.

Asymptotic notation: $o(\cdot)$

- We write $f(x) = o(g(x))$ (*little o*) if for every constant $c > 0$ there is a value x_0 such that $f(x) < cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = x^2$ and $g(x) = x^3$, we have

$$x^2 = o(x^3),$$

because for every $c > 0$, the statement $x^2 < cx^3$ holds for x sufficiently large (just pick $x > 1/c$).

Asymptotic notation: $o(\cdot)$

- We write $f(x) = o(g(x))$ (*little o*) if for every constant $c > 0$ there is a value x_0 such that $f(x) < cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = x^2$ and $g(x) = x^3$, we have

  $$x^2 = o(x^3),$$

  because for every $c > 0$, the statement $x^2 < cx^3$ holds for x sufficiently large (just pick $x > 1/c$).
- What could we write for $x \log x$?

Asymptotic notation: $o(\cdot)$

- We write $f(x) = o(g(x))$ (*little o*) if for every constant $c > 0$ there is a value x_0 such that $f(x) < cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = x^2$ and $g(x) = x^3$, we have
$$x^2 = o(x^3),$$
because for every $c > 0$, the statement $x^2 < cx^3$ holds for x sufficiently large (just pick $x > 1/c$).
- What could we write for $x \log x$?
- We could say:
$$x \log x = o(x^2).$$

Asymptotic notation: $o(\cdot)$

- We write $f(x) = o(g(x))$ (*little o*) if for every constant $c > 0$ there is a value x_0 such that $f(x) < cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = x^2$ and $g(x) = x^3$, we have

$$x^2 = o(x^3),$$

  because for every $c > 0$, the statement $x^2 < cx^3$ holds for x sufficiently large (just pick $x > 1/c$).
- What could we write for $x \log x$?
- We could say:

$$x \log x = o(x^2).$$

- To prove this, note that it's true if for every $c > 0$, we have $x \log x < cx^2$ for x sufficiently large.

Asymptotic notation: $o(\cdot)$

- We write $f(x) = o(g(x))$ (*little o*) if for every constant $c > 0$ there is a value x_0 such that $f(x) < cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = x^2$ and $g(x) = x^3$, we have

$$x^2 = o(x^3),$$

  because for every $c > 0$, the statement $x^2 < cx^3$ holds for x sufficiently large (just pick $x > 1/c$).
- What could we write for $x \log x$?
- We could say:

$$x \log x = o(x^2).$$

- To prove this, note that it's true if for every $c > 0$, we have $x \log x < cx^2$ for x sufficiently large.
- And $x \log x < cx^2$ is equivalent to $(1/c) \log x < x$, which is equivalent to $x^{1/c} < e^x$ and is true for x large enough.

Asymptotic notation: $\Omega(\cdot)$ and $\omega(\cdot)$

Asymptotic notation: $\Omega(\cdot)$ and $\omega(\cdot)$

- We write $f(x) = \Omega(g(x))$ (big Omega) if $g(x) = O(f(x))$ – that is, if there is a constant $c > 0$ and a value x_0 such that $f(x) \geq cg(x)$ whenever $x > x_0$.

Asymptotic notation: $\Omega(\cdot)$ and $\omega(\cdot)$

- We write $f(x) = \Omega(g(x))$ (big Omega) if $g(x) = O(f(x))$ – that is, if there is a constant $c > 0$ and a value x_0 such that $f(x) \geq cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = 2x^2 + 1$ and $g(x) = x^2$, we have

$$2x^2 + 1 = \Omega(x^2)$$

because $2x^2 + 1 > x^2$ (taking $c = 1$).

Asymptotic notation: $\Omega(\cdot)$ and $\omega(\cdot)$

- We write $f(x) = \Omega(g(x))$ (big Omega) if $g(x) = O(f(x))$ – that is, if there is a constant $c > 0$ and a value x_0 such that $f(x) \geq cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = 2x^2 + 1$ and $g(x) = x^2$, we have

$$2x^2 + 1 = \Omega(x^2)$$

  because $2x^2 + 1 > x^2$ (taking $c = 1$).
- We write $f(x) = \omega(g(x))$ (*little omega*) if $g(x) = o(f(x))$ – that is, if for every constant $c > 0$ there is a value x_0 such that $f(x) > cg(x)$ whenever $x > x_0$.

Asymptotic notation: $\Omega(\cdot)$ and $\omega(\cdot)$

- We write $f(x) = \Omega(g(x))$ (big Omega) if $g(x) = O(f(x))$ – that is, if there is a constant $c > 0$ and a value x_0 such that $f(x) \geq cg(x)$ whenever $x > x_0$.
- Example: if $f(x) = 2x^2 + 1$ and $g(x) = x^2$, we have

$$2x^2 + 1 = \Omega(x^2)$$

  because $2x^2 + 1 > x^2$ (taking $c = 1$).
- We write $f(x) = \omega(g(x))$ (*little omega*) if $g(x) = o(f(x))$ – that is, if for every constant $c > 0$ there is a value x_0 such that $f(x) > cg(x)$ whenever $x > x_0$.
- Example:

$$x^2 \log x = \omega(x^2).$$

Asymptotic notation: $\Theta(\cdot)$

Asymptotic notation: $\Theta(\cdot)$

- We write $f(x) = \Theta(g(x))$ if both $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.

Asymptotic notation: $\Theta(\cdot)$

- We write $f(x) = \Theta(g(x))$ if both $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.
- Example:

$$2x^2 + 1 = \Theta(x^2)$$

Asymptotic notation: $\Theta(\cdot)$

- We write $f(x) = \Theta(g(x))$ if both $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.
- Example:

$$2x^2 + 1 = \Theta(x^2)$$

Asymptotic notation: $\Theta(\cdot)$

- We write $f(x) = \Theta(g(x))$ if both $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.
- Example:
$$2x^2 + 1 = \Theta(x^2)$$
- What about if both $f(x) = o(g(x))$ and $f(x) = \omega(g(x))$ are true?

Asymptotic notation: $\Theta(\cdot)$

- We write $f(x) = \Theta(g(x))$ if both $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.
- Example:

$$2x^2 + 1 = \Theta(x^2)$$

- What about if both $f(x) = o(g(x))$ and $f(x) = \omega(g(x))$ are true?
- This would mean that for every c, $f(x) < cg(x)$ and $f(x) > cg(x)$ for x sufficiently large.

Asymptotic notation: $\Theta(\cdot)$

- We write $f(x) = \Theta(g(x))$ if both $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.
- Example:

$$2x^2 + 1 = \Theta(x^2)$$

- What about if both $f(x) = o(g(x))$ and $f(x) = \omega(g(x))$ are true?
- This would mean that for every c, $f(x) < cg(x)$ and $f(x) > cg(x)$ for x sufficiently large.
- It isn't possible to have both of these be true simultaneously!

Examples

Examples

- Anything later in this sequence is $\omega(\cdot)$ of anything earlier:

$$\log x, x, x \log x, x^2, x^3, x^4, \ldots, e^x, e^{2x}, \ldots$$

Examples

- Anything later in this sequence is $\omega(\cdot)$ of anything earlier:

$$\log x, x, x \log x, x^2, x^3, x^4, \ldots, e^x, e^{2x}, \ldots$$

- Why is $(x+1)^{10} = \Theta(x^{10})$?

Examples

- Anything later in this sequence is $\omega(\cdot)$ of anything earlier:

$$\log x, x, x \log x, x^2, x^3, x^4, \ldots, e^x, e^{2x}, \ldots$$

- Why is $(x+1)^{10} = \Theta(x^{10})$?
- Expanding it out, we have

$$(x+1)^{10} = x^{10} + 10x^9 + \binom{10}{2} x^8 + \cdots + 10x + 1.$$

Examples

- Anything later in this sequence is $\omega(\cdot)$ of anything earlier:

$$\log x, x, x \log x, x^2, x^3, x^4, \ldots, e^x, e^{2x}, \ldots$$

- Why is $(x+1)^{10} = \Theta(x^{10})$?
- Expanding it out, we have

$$(x+1)^{10} = x^{10} + 10x^9 + \binom{10}{2} x^8 + \cdots + 10x + 1.$$

- So for example $2x^{10} > (x+1)^{10}$ for x sufficiently large, so $(x+1)^{10} = O(x^{10})$.

Examples

- Anything later in this sequence is $\omega(\cdot)$ of anything earlier:

$$\log x, x, x\log x, x^2, x^3, x^4, \ldots, e^x, e^{2x}, \ldots$$

- Why is $(x+1)^{10} = \Theta(x^{10})$?
- Expanding it out, we have

$$(x+1)^{10} = x^{10} + 10x^9 + \binom{10}{2}x^8 + \cdots + 10x + 1.$$

- So for example $2x^{10} > (x+1)^{10}$ for x sufficiently large, so $(x+1)^{10} = O(x^{10})$.
- And $(x+1)^{10} = \Omega(x^{10})$ is clear, since $(x+1)^{10} > x^{10}$.

Simple operations

Simple operations

- If $f(x) = O(g(x))$, then $cf(x) = O(g(x))$ for constant $c > 0$.

Simple operations

- If $f(x) = O(g(x))$, then $cf(x) = O(g(x))$ for constant $c > 0$.
- If $f(x) = O(h(x))$ and $g(x) = O(h(x))$, then

$$f(x) + g(x) = O(h(x)).$$

Simple operations

- If $f(x) = O(g(x))$, then $cf(x) = O(g(x))$ for constant $c > 0$.
- If $f(x) = O(h(x))$ and $g(x) = O(h(x))$, then

$$f(x) + g(x) = O(h(x)).$$

- Same statements also hold for $o(\cdot)$, $\Omega(\cdot)$, $\omega(\cdot)$, and $\Theta(\cdot)$.

Why asymptotic notation is useful

Why asymptotic notation is useful

- We will often use asymptotic notation for assessing the time (or space) required for an algorithm, as a function of input size.

Why asymptotic notation is useful

- We will often use asymptotic notation for assessing the time (or space) required for an algorithm, as a function of input size.
- That's because the constants are often less important than how the time scales with the input, e.g. $\Theta(n^2)$ is generally much worse than $\Theta(n)$.

Why asymptotic notation is useful

- We will often use asymptotic notation for assessing the time (or space) required for an algorithm, as a function of input size.
- That's because the constants are often less important than how the time scales with the input, e.g. $\Theta(n^2)$ is generally much worse than $\Theta(n)$.
- Also, constants often depend on the units we are using, the particular hardware, etc.

Why asymptotic notation is useful

- We will often use asymptotic notation for assessing the time (or space) required for an algorithm, as a function of input size.
- That's because the constants are often less important than how the time scales with the input, e.g. $\Theta(n^2)$ is generally much worse than $\Theta(n)$.
- Also, constants often depend on the units we are using, the particular hardware, etc.
- Asymptotic notation means we don't have to worry about that.

Time complexity of algorithms

Time complexity of algorithms

- The *time complexity* of an algorithm quantifies how much time it takes to run the algorithm.

Time complexity of algorithms

- The *time complexity* of an algorithm quantifies how much time it takes to run the algorithm.
- It is possible to think about *worst-case* time (maximum amount of time it takes to run).

Time complexity of algorithms

- The *time complexity* of an algorithm quantifies how much time it takes to run the algorithm.
- It is possible to think about *worst-case* time (maximum amount of time it takes to run).
- Or *average-case* time (average amount of time it takes to run over all possible inputs).

Sorting

Problem: Given a list L of n numbers, all of them different, put them into increasing order.

Sorting

Problem: Given a list L of n numbers, all of them different, put them into increasing order.

- First cut at how to do this: **Insertion sort** (probably what you do if you are sorting e.g. a deck of cards).

Sorting

Problem: Given a list L of n numbers, all of them different, put them into increasing order.

- First cut at how to do this: **Insertion sort** (probably what you do if you are sorting e.g. a deck of cards).
- Create a new list L' one number at a time.

Sorting

Problem: Given a list L of n numbers, all of them different, put them into increasing order.

- First cut at how to do this: **Insertion sort** (probably what you do if you are sorting e.g. a deck of cards).
- Create a new list L' one number at a time.
- For every number in L, work out where it should be in L' and add it there.

Sorting

Problem: Given a list L of n numbers, all of them different, put them into increasing order.

- First cut at how to do this: **Insertion sort** (probably what you do if you are sorting e.g. a deck of cards).
- Create a new list L' one number at a time.
- For every number in L, work out where it should be in L' and add it there.
- Example:

$$5, 2, 6, 1, 4, 3 \Rightarrow$$

Sorting

Problem: Given a list L of n numbers, all of them different, put them into increasing order.

- First cut at how to do this: **Insertion sort** (probably what you do if you are sorting e.g. a deck of cards).
- Create a new list L' one number at a time.
- For every number in L, work out where it should be in L' and add it there.
- Example:

$$\mathbf{5}, 2, 6, 1, 4, 3 \Rightarrow 5$$

Sorting

Problem: Given a list L of n numbers, all of them different, put them into increasing order.

- First cut at how to do this: **Insertion sort** (probably what you do if you are sorting e.g. a deck of cards).
- Create a new list L' one number at a time.
- For every number in L, work out where it should be in L' and add it there.
- Example:

$$5, \mathbf{2}, 6, 1, 4, 3 \Rightarrow 2, 5$$

Sorting

Problem: Given a list L of n numbers, all of them different, put them into increasing order.

- First cut at how to do this: **Insertion sort** (probably what you do if you are sorting e.g. a deck of cards).
- Create a new list L' one number at a time.
- For every number in L, work out where it should be in L' and add it there.
- Example:

$$5, 2, \mathbf{6}, 1, 4, 3 \Rightarrow 2, 5, 6$$

Sorting

Problem: Given a list L of n numbers, all of them different, put them into increasing order.

- First cut at how to do this: **Insertion sort** (probably what you do if you are sorting e.g. a deck of cards).
- Create a new list L' one number at a time.
- For every number in L, work out where it should be in L' and add it there.
- Example:

$$5, 2, 6, \mathbf{1}, 4, 3 \Rightarrow 1, 2, 5, 6$$

Sorting

Problem: Given a list L of n numbers, all of them different, put them into increasing order.

- First cut at how to do this: **Insertion sort** (probably what you do if you are sorting e.g. a deck of cards).
- Create a new list L' one number at a time.
- For every number in L, work out where it should be in L' and add it there.
- Example:

$$5, 2, 6, 1, \mathbf{4}, 3 \Rightarrow 1, 2, 4, 5, 6$$

Sorting

Problem: Given a list L of n numbers, all of them different, put them into increasing order.

- First cut at how to do this: **Insertion sort** (probably what you do if you are sorting e.g. a deck of cards).
- Create a new list L' one number at a time.
- For every number in L, work out where it should be in L' and add it there.
- Example:

$$5, 2, 6, 1, 4, \mathbf{3} \Rightarrow 1, 2, 3, 4, 5, 6$$

Insertion sort

Insertion sort

- Let's work out how long this takes.

Insertion sort

- Let's work out how long this takes.
- Suppose we are taking the kth element of L and inserting into L'.

Insertion sort

- Let's work out how long this takes.
- Suppose we are taking the kth element of L and inserting into L'.
- To insert it into L', we can first move it to the end of L' and then iteratively swap any elements of L' lower than it:

Insertion sort

- Let's work out how long this takes.
- Suppose we are taking the kth element of L and inserting into L'.
- To insert it into L', we can first move it to the end of L' and then iteratively swap any elements of L' lower than it:

$$1, 2, 4, 5, 6, \mathbf{3}$$

Insertion sort

- Let's work out how long this takes.
- Suppose we are taking the kth element of L and inserting into L'.
- To insert it into L', we can first move it to the end of L' and then iteratively swap any elements of L' lower than it:

$$1, 2, 4, 5, \mathbf{3}, 6$$

Insertion sort

- Let's work out how long this takes.
- Suppose we are taking the kth element of L and inserting into L'.
- To insert it into L', we can first move it to the end of L' and then iteratively swap any elements of L' lower than it:

$$1, 2, 4, \mathbf{3}, 5, 6$$

Insertion sort

- Let's work out how long this takes.
- Suppose we are taking the kth element of L and inserting into L'.
- To insert it into L', we can first move it to the end of L' and then iteratively swap any elements of L' lower than it:

$$1, 2, \mathbf{3}, 4, 5, 6$$

Insertion sort

- Let's work out how long this takes.
- Suppose we are taking the kth element of L and inserting into L'.
- To insert it into L', we can first move it to the end of L' and then iteratively swap any elements of L' lower than it:

$$1, 2, 3, 4, 5, 6$$

Insertion sort

- Let's work out how long this takes.
- Suppose we are taking the kth element of L and inserting into L'.
- To insert it into L', we can first move it to the end of L' and then iteratively swap any elements of L' lower than it:

$$1, 2, 3, 4, 5, 6$$

- This takes m_k steps, where m_k is the number of spots from the end of L' that we have to move.

Insertion sort

- Let's work out how long this takes.
- Suppose we are taking the kth element of L and inserting into L'.
- To insert it into L', we can first move it to the end of L' and then iteratively swap any elements of L' lower than it:

$$1, 2, 3, 4, 5, 6$$

- This takes m_k steps, where m_k is the number of spots from the end of L' that we have to move.
- The overall time of the whole algorithm is then $m = \sum_{k=1}^{n} m_k$.

Insertion sort

- Let's work out how long this takes.
- Suppose we are taking the kth element of L and inserting into L'.
- To insert it into L', we can first move it to the end of L' and then iteratively swap any elements of L' lower than it:

$$1, 2, 3, 4, 5, 6$$

- This takes m_k steps, where m_k is the number of spots from the end of L' that we have to move.
- The overall time of the whole algorithm is then $m = \sum_{k=1}^{n} m_k$.
- What is the worst-case time complexity of the algorithm?

Insertion sort

- Let's work out how long this takes.
- Suppose we are taking the kth element of L and inserting into L'.
- To insert it into L', we can first move it to the end of L' and then iteratively swap any elements of L' lower than it:

$$1, 2, 3, 4, 5, 6$$

- This takes m_k steps, where m_k is the number of spots from the end of L' that we have to move.
- The overall time of the whole algorithm is then $m = \sum_{k=1}^{n} m_k$.
- What is the worst-case time complexity of the algorithm?
- In the worst case, $m_k = k$ for each k (the list starts out in reverse order), so we have

$$m = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2).$$

Insertion sort

Insertion sort

- What is the average-case time complexity of the algorithm?

Insertion sort

- What is the average-case time complexity of the algorithm?
- We are looking for $\mathbb{E}[m]$, and we know $m = \sum_{k=1}^{n} m_k$. What can we use?

Insertion sort

- What is the average-case time complexity of the algorithm?
- We are looking for $\mathbb{E}[m]$, and we know $m = \sum_{k=1}^{n} m_k$. What can we use?
- We can use linearity of expectation:

$$\mathbb{E}[m] = \sum_{k=1}^{n} \mathbb{E}[m_k].$$

Insertion sort

- What is the average-case time complexity of the algorithm?
- We are looking for $\mathbb{E}[m]$, and we know $m = \sum_{k=1}^{n} m_k$. What can we use?
- We can use linearity of expectation:

$$\mathbb{E}[m] = \sum_{k=1}^{n} \mathbb{E}[m_k].$$

- What is $\mathbb{E}[m_k]$?

Insertion sort

- What is the average-case time complexity of the algorithm?
- We are looking for $\mathbb{E}[m]$, and we know $m = \sum_{k=1}^{n} m_k$. What can we use?
- We can use linearity of expectation:

$$\mathbb{E}[m] = \sum_{k=1}^{n} \mathbb{E}[m_k].$$

- What is $\mathbb{E}[m_k]$?
- Since the first k elements of L can be in any order, m_k is equally likely to be any of $1, 2, 3, \ldots, k$.

Insertion sort

- What is the average-case time complexity of the algorithm?
- We are looking for $\mathbb{E}[m]$, and we know $m = \sum_{k=1}^{n} m_k$. What can we use?
- We can use linearity of expectation:

$$\mathbb{E}[m] = \sum_{k=1}^{n} \mathbb{E}[m_k].$$

- What is $\mathbb{E}[m_k]$?
- Since the first k elements of L can be in any order, m_k is equally likely to be any of $1, 2, 3, \ldots, k$.
- Therefore, $\mathbb{E}[m_k] = k/2$.

Insertion sort

- What is the average-case time complexity of the algorithm?
- We are looking for $\mathbb{E}[m]$, and we know $m = \sum_{k=1}^{n} m_k$. What can we use?
- We can use linearity of expectation:

$$\mathbb{E}[m] = \sum_{k=1}^{n} \mathbb{E}[m_k].$$

- What is $\mathbb{E}[m_k]$?
- Since the first k elements of L can be in any order, m_k is equally likely to be any of $1, 2, 3, \ldots, k$.
- Therefore, $\mathbb{E}[m_k] = k/2$.
- Overall, we have:

$$\mathbb{E}[m] = \sum_{k=1}^{n} k/2 = \frac{n(n+1)}{4} = \Theta(n^2).$$

Insertion sort

- What is the average-case time complexity of the algorithm?
- We are looking for $\mathbb{E}[m]$, and we know $m = \sum_{k=1}^{n} m_k$. What can we use?
- We can use linearity of expectation:

$$\mathbb{E}[m] = \sum_{k=1}^{n} \mathbb{E}[m_k].$$

- What is $\mathbb{E}[m_k]$?
- Since the first k elements of L can be in any order, m_k is equally likely to be any of $1, 2, 3, \ldots, k$.
- Therefore, $\mathbb{E}[m_k] = k/2$.
- Overall, we have:

$$\mathbb{E}[m] = \sum_{k=1}^{n} k/2 = \frac{n(n+1)}{4} = \Theta(n^2).$$

- In this case, both worst-case and average-case times are $\Theta(n^2)$.

Merge sort

Merge sort

- We can also use the *dynamic programming* paradigm: Divide the problem into multiple smaller problems, and solve them separately.

Merge sort

- We can also use the *dynamic programming* paradigm: Divide the problem into multiple smaller problems, and solve them separately.
- For sorting, this gives us the **merge sort** algorithm.

Merge sort

- We can also use the *dynamic programming* paradigm: Divide the problem into multiple smaller problems, and solve them separately.
- For sorting, this gives us the **merge sort** algorithm.
- 1) Divide the full list L into lists L_1 and L_2.

Merge sort

- We can also use the *dynamic programming* paradigm: Divide the problem into multiple smaller problems, and solve them separately.
- For sorting, this gives us the **merge sort** algorithm.
- 1) Divide the full list L into lists L_1 and L_2.
- 2) Sort each of L_1 and L_2 recursively with the same algorithm.

Merge sort

- We can also use the *dynamic programming* paradigm: Divide the problem into multiple smaller problems, and solve them separately.
- For sorting, this gives us the **merge sort** algorithm.
- 1) Divide the full list L into lists L_1 and L_2.
- 2) Sort each of L_1 and L_2 recursively with the same algorithm.
- 3) Merge the sorted lists L_1 and L_2.

Merge sort

- We can also use the *dynamic programming* paradigm: Divide the problem into multiple smaller problems, and solve them separately.
- For sorting, this gives us the **merge sort** algorithm.
- 1) Divide the full list L into lists L_1 and L_2.
- 2) Sort each of L_1 and L_2 recursively with the same algorithm.
- 3) Merge the sorted lists L_1 and L_2.
- Example:

$$\begin{aligned}
\text{sort}([5,2,6,1,4,3]) &= \text{merge}(\text{sort}([5,2,6]), \text{sort}([1,4,3])) \\
&= \text{merge}(\text{merge}(\text{sort}([5,2]), \text{sort}([6])), \\
&\quad\ \text{merge}(\text{sort}([1,4]), \text{sort}([3]))) \\
&= \text{merge}(\text{merge}([2,5],[6]), \text{merge}([1,4],[3])) \\
&= \text{merge}([2,5,6],[1,3,4]) \\
&= [1,2,3,4,5,6]
\end{aligned}$$

Merge sort

Merge sort

- How do we do the merging?

Merge sort

- How do we do the merging?
- Suppose we have two sorted decks of cards.

Merge sort

- How do we do the merging?
- Suppose we have two sorted decks of cards.
- We can flip the top two cards (min of each deck).

Merge sort

- How do we do the merging?
- Suppose we have two sorted decks of cards.
- We can flip the top two cards (min of each deck).
- Take the min of those two, flip the card below.

Merge sort

- How do we do the merging?
- Suppose we have two sorted decks of cards.
- We can flip the top two cards (min of each deck).
- Take the min of those two, flip the card below.
- Continue, taking the min from the two.

Merge sort

- How do we do the merging?
- Suppose we have two sorted decks of cards.
- We can flip the top two cards (min of each deck).
- Take the min of those two, flip the card below.
- Continue, taking the min from the two.
- Stop when one deck is empty.

Merge sort

- How do we do the merging?
- Suppose we have two sorted decks of cards.
- We can flip the top two cards (min of each deck).
- Take the min of those two, flip the card below.
- Continue, taking the min from the two.
- Stop when one deck is empty.
- If L_1 and L_2 have lengths n_1 and n_2, how long does this take (worst-case)?

Merge sort

- How do we do the merging?
- Suppose we have two sorted decks of cards.
- We can flip the top two cards (min of each deck).
- Take the min of those two, flip the card below.
- Continue, taking the min from the two.
- Stop when one deck is empty.
- If L_1 and L_2 have lengths n_1 and n_2, how long does this take (worst-case)?
- Worst case, we have to flip all cards, so $c(n_1 + n_2)$ for some c.

Merge sort

Merge sort

- What is the worst-case time complexity for the whole algorithm?

Merge sort

- What is the worst-case time complexity for the whole algorithm?
- Let's set $f(n)$ to be the worst-case time for lists L of length n.

Merge sort

- What is the worst-case time complexity for the whole algorithm?
- Let's set $f(n)$ to be the worst-case time for lists L of length n.
- We need to sort L_1 and L_2 and merge them.

Merge sort

- What is the worst-case time complexity for the whole algorithm?
- Let's set $f(n)$ to be the worst-case time for lists L of length n.
- We need to sort L_1 and L_2 and merge them.
- If n is even, that gives us:

$$f(n) = f(n/2) + f(n/2) + c(n/2 + n/2) = 2f(n/2) + cn.$$

Merge sort

- What is the worst-case time complexity for the whole algorithm?
- Let's set $f(n)$ to be the worst-case time for lists L of length n.
- We need to sort L_1 and L_2 and merge them.
- If n is even, that gives us:

$$f(n) = f(n/2) + f(n/2) + c(n/2 + n/2) = 2f(n/2) + cn.$$

- The analysis is simpler if $n = 2^k$:

$$\begin{aligned} f(2^k) &= 2f(2^{k-1}) + 2^k c \\ &= 2(2f(2^{k-2}) + 2^{k-1}c) + 2^k c \\ &= 4f(2^{k-2}) + 2 \cdot 2^k c \\ &= 4(2f(2^{k-3}) + 2^{k-2}c) + 2 \cdot 2^k c \\ &= 8f(2^{k-3}) + 3 \cdot 2^k c \\ &= \cdots \\ &= 2^k f(1) + k \cdot 2^k c. \end{aligned}$$

Merge sort

- What is the worst-case time complexity for the whole algorithm?
- Let's set $f(n)$ to be the worst-case time for lists L of length n.
- We need to sort L_1 and L_2 and merge them.
- If n is even, that gives us:

$$f(n) = f(n/2) + f(n/2) + c(n/2 + n/2) = 2f(n/2) + cn.$$

- The analysis is simpler if $n = 2^k$:

$$f(2^k) = 2^k f(1) + k \cdot 2^k c.$$

Merge sort

- What is the worst-case time complexity for the whole algorithm?
- Let's set $f(n)$ to be the worst-case time for lists L of length n.
- We need to sort L_1 and L_2 and merge them.
- If n is even, that gives us:

$$f(n) = f(n/2) + f(n/2) + c(n/2 + n/2) = 2f(n/2) + cn.$$

- The analysis is simpler if $n = 2^k$:

$$f(2^k) = 2^k f(1) + k \cdot 2^k c.$$

- From this:

$$f(2^k) = k \cdot 2^k c = \Theta(k2^k) = \Theta(n \log n).$$

Merge sort

- What is the worst-case time complexity for the whole algorithm?
- Let's set $f(n)$ to be the worst-case time for lists L of length n.
- We need to sort L_1 and L_2 and merge them.
- If n is even, that gives us:

$$f(n) = f(n/2) + f(n/2) + c(n/2 + n/2) = 2f(n/2) + cn.$$

- The analysis is simpler if $n = 2^k$:

$$f(2^k) = 2^k f(1) + k \cdot 2^k c.$$

- From this:

$$f(2^k) = k \cdot 2^k c = \Theta(k2^k) = \Theta(n \log n).$$

- Similar analysis for $n \neq 2^k$, get $f(n) = \Theta(n \log n)$.

Merge sort

- What is the worst-case time complexity for the whole algorithm?
- Let's set $f(n)$ to be the worst-case time for lists L of length n.
- We need to sort L_1 and L_2 and merge them.
- If n is even, that gives us:

$$f(n) = f(n/2) + f(n/2) + c(n/2 + n/2) = 2f(n/2) + cn.$$

- The analysis is simpler if $n = 2^k$:

$$f(2^k) = 2^k f(1) + k \cdot 2^k c.$$

- From this:

$$f(2^k) = k \cdot 2^k c = \Theta(k2^k) = \Theta(n \log n).$$

- Similar analysis for $n \neq 2^k$, get $f(n) = \Theta(n \log n)$.
- Can show that average-case time is also $\Theta(n \log n)$.

Problem

Prove that for integers $n \geq 2$, we have:

$$\frac{1}{2} + \cdots + \frac{1}{n} \leq \log n \leq 1 + \frac{1}{2} + \cdots + \frac{1}{n-1}.$$

Problem

Prove that for integers $n \geq 2$, we have:

$$\frac{1}{2} + \cdots + \frac{1}{n} \leq \log n \leq 1 + \frac{1}{2} + \cdots + \frac{1}{n-1}.$$

- The area under the curve $y = 1/x$ between $x = 1$ and $x = n$ is

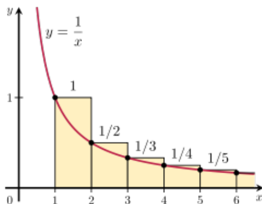
$$\int_1^n 1/x \, dx = \log(n) - \log(1) = \log(n).$$

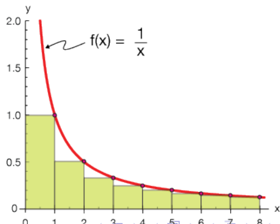
Problem

Prove that for integers $n \geq 2$, we have:

$$\frac{1}{2} + \cdots + \frac{1}{n} \leq \log n \leq 1 + \frac{1}{2} + \cdots + \frac{1}{n-1}.$$

- The area under the curve $y = 1/x$ between $x = 1$ and $x = n$ is
$$\int_1^n 1/x \, dx = \log(n) - \log(1) = \log(n).$$
- But this area is upper-bounded by the sum of areas of boxes $1 + 1/2 + \cdots + 1/(n-1)$ and is lower-bounded by $1/2 + 1/3 + \cdots + 1/n$.





Quicksort

Quicksort

- **Quicksort** is another dynamic programming approach.

Quicksort

- **Quicksort** is another dynamic programming approach.
- 1) Take an element x from somewhere in the list L.

Quicksort

- **Quicksort** is another dynamic programming approach.
- 1) Take an element x from somewhere in the list L.
- 2) Go through L one-by-one and *pivot* using x: that is, put anything lower than x in L_1, anything higher than x in L_2.

Quicksort

- **Quicksort** is another dynamic programming approach.
- 1) Take an element x from somewhere in the list L.
- 2) Go through L one-by-one and *pivot* using x: that is, put anything lower than x in L_1, anything higher than x in L_2.
- 3) Sort L_1 and L_2 recursively.

Quicksort

- **Quicksort** is another dynamic programming approach.
- 1) Take an element x from somewhere in the list L.
- 2) Go through L one-by-one and *pivot* using x: that is, put anything lower than x in L_1, anything higher than x in L_2.
- 3) Sort L_1 and L_2 recursively.
- 4) Combine L_1, x, L_2, in that order.

Quicksort

- **Quicksort** is another dynamic programming approach.
- 1) Take an element x from somewhere in the list L.
- 2) Go through L one-by-one and *pivot* using x: that is, put anything lower than x in L_1, anything higher than x in L_2.
- 3) Sort L_1 and L_2 recursively.
- 4) Combine L_1, x, L_2, in that order.
- Example of pivoting around 4:

$$\text{sort}([5, 2, 6, 1, 4, 3]) = \text{sort}([2, 1, 3]) + [4] + \text{sort}([5, 6]).$$

Quicksort

- **Quicksort** is another dynamic programming approach.
- 1) Take an element x from somewhere in the list L.
- 2) Go through L one-by-one and *pivot* using x: that is, put anything lower than x in L_1, anything higher than x in L_2.
- 3) Sort L_1 and L_2 recursively.
- 4) Combine L_1, x, L_2, in that order.
- Example of pivoting around 4:

$$\text{sort}([5,2,6,1,4,3]) = \text{sort}([2,1,3]) + [4] + \text{sort}([5,6]).$$

- Many ways to choose the pivot x.

Quicksort

- **Quicksort** is another dynamic programming approach.
- 1) Take an element x from somewhere in the list L.
- 2) Go through L one-by-one and *pivot* using x: that is, put anything lower than x in L_1, anything higher than x in L_2.
- 3) Sort L_1 and L_2 recursively.
- 4) Combine L_1, x, L_2, in that order.
- Example of pivoting around 4:

$$\text{sort}([5,2,6,1,4,3]) = \text{sort}([2,1,3]) + [4] + \text{sort}([5,6]).$$

- Many ways to choose the pivot x.
- Let's assume it's random.

Quicksort

Quicksort

- How long does each pivot take?

Quicksort

- How long does each pivot take?
- We have to compare x to every element in the list, so cn time.

Quicksort

- How long does each pivot take?
- We have to compare x to every element in the list, so cn time.
- So if the total time to sort list L is $f(L)$, then we have:

$$f(L) = f(L_1) + f(L_2) + cn.$$

Quicksort

- How long does each pivot take?
- We have to compare x to every element in the list, so cn time.
- So if the total time to sort list L is $f(L)$, then we have:

$$f(L) = f(L_1) + f(L_2) + cn.$$

- What is the best pivot we could pick?

Quicksort

- How long does each pivot take?
- We have to compare x to every element in the list, so cn time.
- So if the total time to sort list L is $f(L)$, then we have:

$$f(L) = f(L_1) + f(L_2) + cn.$$

- The best pivot is intuitively the midway point of the list.

Quicksort

- How long does each pivot take?
- We have to compare x to every element in the list, so cn time.
- So if the total time to sort list L is $f(L)$, then we have:

$$f(L) = f(L_1) + f(L_2) + cn.$$

- The best pivot is intuitively the midway point of the list.
- Using $f(n)$ for the time on a list of length n, if we could always pick the midway point we would have:

$$f(n) = f(n/2) + f(n/2) + cn,$$

  which we saw means $f(n) = \Theta(n \log n)$.

Quicksort

- How long does each pivot take?
- We have to compare x to every element in the list, so cn time.
- So if the total time to sort list L is $f(L)$, then we have:

$$f(L) = f(L_1) + f(L_2) + cn.$$

- The best pivot is intuitively the midway point of the list.
- Using $f(n)$ for the time on a list of length n, if we could always pick the midway point we would have:

$$f(n) = f(n/2) + f(n/2) + cn,$$

  which we saw means $f(n) = \Theta(n \log n)$.
- However, we do not know what the elements of the list are in advance, and computing the midpoint takes time.

Quicksort

- How long does each pivot take?
- We have to compare x to every element in the list, so cn time.
- So if the total time to sort list L is $f(L)$, then we have:

$$f(L) = f(L_1) + f(L_2) + cn.$$

- What is the worst pivot we could pick?

Quicksort

- How long does each pivot take?
- We have to compare x to every element in the list, so cn time.
- So if the total time to sort list L is $f(L)$, then we have:

$$f(L) = f(L_1) + f(L_2) + cn.$$

- The worst pivot is intuitively the one that does the worst job of dividing the list into two, so the greatest (or smallest element).

Quicksort

- How long does each pivot take?
- We have to compare x to every element in the list, so cn time.
- So if the total time to sort list L is $f(L)$, then we have:

$$f(L) = f(L_1) + f(L_2) + cn.$$

- The worst pivot is intuitively the one that does the worst job of dividing the list into two, so the greatest (or smallest element).
- (We won't prove this, but it is straightforward to prove.)

Quicksort

- How long does each pivot take?
- We have to compare x to every element in the list, so cn time.
- So if the total time to sort list L is $f(L)$, then we have:

$$f(L) = f(L_1) + f(L_2) + cn.$$

- The worst pivot is intuitively the one that does the worst job of dividing the list into two, so the greatest (or smallest element).
- (We won't prove this, but it is straightforward to prove.)
- In this case, we have:

$$\begin{aligned} f(n) &= f(n-1) + f(1) + cn \\ &= f(n-1) + cn \\ &= cn + c(n-1) + \ldots + c \\ &= c(n + (n-1) + \ldots + 1) \\ &= cn(n-1)/2 \\ &= \Theta(n^2). \end{aligned}$$

Quicksort

Quicksort

- Therefore, it seems the best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.

Quicksort

- Therefore, it seems the best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.
- What do we think is the average-case time?

Quicksort

- Therefore, it seems the best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.
- What do we think is the average-case time?
- The time of the algorithm is cX, where X is the number of comparisons between two elements to see which is bigger.

Quicksort

- Therefore, it seems the best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.
- What do we think is the average-case time?
- The time of the algorithm is cX, where X is the number of comparisons between two elements to see which is bigger.
- Suppose the list L contains the numbers $z_1 < z_2 < \cdots < z_n$ in some order.

Quicksort

- Therefore, it seems the best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.
- What do we think is the average-case time?
- The time of the algorithm is cX, where X is the number of comparisons between two elements to see which is bigger.
- Suppose the list L contains the numbers $z_1 < z_2 < \cdots < z_n$ in some order.
- We have $X = \sum_{i<j} X_{ij}$, where X_{ij} is the indicator variable for whether numbers z_i and z_j are compared.

Quicksort

- Therefore, it seems the best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.
- What do we think is the average-case time?
- The time of the algorithm is cX, where X is the number of comparisons between two elements to see which is bigger.
- Suppose the list L contains the numbers $z_1 < z_2 < \cdots < z_n$ in some order.
- We have $X = \sum_{i<j} X_{ij}$, where X_{ij} is the indicator variable for whether numbers z_i and z_j are compared.
- By linearity of expectation:

$$\mathbb{E}[X] = \sum_{i<j} \mathbb{E}[X_{ij}] = \sum_{i<j} p(\text{comparing } z_i \text{ and } z_j)$$

Quicksort

Quicksort

- Now, let's think about when z_i and z_j are compared.

Quicksort

- Now, let's think about when z_i and z_j are compared.
- Example:

$$\text{sort}([5, 2, 6, 1, 4, 3]) = \text{sort}([2, 1, 3]) + [4] + \text{sort}([5, 6]).$$

Quicksort

- Now, let's think about when z_i and z_j are compared.
- Example:

$$\text{sort}([5,2,6,1,4,3]) = \text{sort}([2,1,3]) + [4] + \text{sort}([5,6]).$$

- z_i and z_j are not compared if we first pick a pivot between them.

Quicksort

- Now, let's think about when z_i and z_j are compared.
- Example:

$$\text{sort}([5,2,6,1,4,3]) = \text{sort}([2,1,3]) + [4] + \text{sort}([5,6]).$$

- z_i and z_j are not compared if we first pick a pivot between them.
- So they are compared exactly when we pick z_i or z_j as a pivot *before* picking any of the other numbers between z_i and z_j.

Quicksort

- Now, let's think about when z_i and z_j are compared.
- Example:

$$\text{sort}([5,2,6,1,4,3]) = \text{sort}([2,1,3]) + [4] + \text{sort}([5,6]).$$

- z_i and z_j are not compared if we first pick a pivot between them.
- So they are compared exactly when we pick z_i or z_j as a pivot *before* picking any of the other numbers between z_i and z_j.
- Therefore, we have:

$$\begin{aligned} & p(\text{comparing } z_i \text{ and } z_j) \\ = \; & p(\text{picking } z_i \text{ first out of the nums between } z_i \text{ and } z_j) \\ & + p(\text{picking } z_j \text{ first out of the nums between } z_i \text{ and } z_j) \\ = \; & \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ = \; & \frac{2}{j-i+1}. \end{aligned}$$

Quicksort

Quicksort

- So we have:

$$
\begin{aligned}
\mathbb{E}[X] &= \sum_{i<j} \mathbb{E}[X_{ij}] \\
&= \sum_{i<j} \frac{2}{j-i+1} \\
&= 2\sum_{i} \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n-i+1} \right) \\
&\leq 2n \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} \right) \\
&= O(n \log n)
\end{aligned}
$$

Quicksort

- So we have:

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i<j} \mathbb{E}[X_{ij}] \\
&= \sum_{i<j} \frac{2}{j-i+1} \\
&= 2\sum_{i} \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n-i+1} \right) \\
&\leq 2n \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} \right) \\
&= O(n \log n)
\end{aligned}$$

- Where we used $\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} = O(\log n)$.

Quicksort

- So we have:

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i<j} \mathbb{E}[X_{ij}] \\ &= \sum_{i<j} \frac{2}{j-i+1} \\ &= 2\sum_{i} \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n-i+1}\right) \\ &\leq 2n\left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}\right) \\ &= O(n \log n)\end{aligned}$$

- Where we used $\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} = O(\log n)$.
- Since best-case running time is $\Theta(n \log n)$, the average running time is not just $O(n \log n)$, it is $\Theta(n \log n)$.

Next time!

Heaps