Additional Note for Algorithms and Data Structures

THORE HUSFELDT

This note includes a careful development of tilde and Landau notation.

This note covers the technique of amortized analysis, as a supplement to [SW], the textbook *Algorithms, 4th ed.* by Sedgewick and Wayne.

It contains a complete answer to [SW, exercise 1.4.32] thus completing the proof of proposition E in [SW, section 1.4].

1 ASYMPTOTIC NOTATION

This note is about notation for the growth of functions encountered in the analysis of algortithms, sometimes called asymptotic[1] notation.

1.1 Notation and use

We begin by introducing notation, intuition, and rules for use. Precise definitions and explanation are given later.

First, though: Why are we doing this at all? We need a toolset to compare functions, because we want to compare the running times of algorithms, which are given as functions (typically of the input size). We both want to be able to say "f is roughly equal to g," and "f is bounded by g" for a certain granularity of "roughly" that is both precise enough to be interesting and coarse enough to be useful.

Note that we *do* already a definition for have "precisely equal," which is typically written as $f = g$ and defined as

$$f(x) = g(x) \qquad \text{for all } x\,.$$

That was easy to define, but it is also quite useless for us; if $f(n) = n$ and $g(n) = n + \frac{1}{1000}$ then $f \neq g$. Similarly we *do* already a definition for "bounded by," typically written as $f \leq g$ and defined as

$$f(x) \leq g(x) \qquad \text{for all } x\,.$$

This is also quite useless: if $f(n) = n^2$ and $g(n) = n^3$ then $f \nleq g$ because $(-1)^2 \nleq (-1)^3$. (But we really want to be able to say that a quadratic-time algorithm is better than a cubic-time one.)

1.1.1 *Asymptotic equality: tilde.*

The notation $f(x) \sim g(x)$, sometimes abbreviated to $f \sim g$, is a "fuzzy version of $f = g$." Informally, $f(x) \sim g(x)$ if $f(x)/g(x)$ approaches 1 as x grows. Some usage examples are

$$n + 1 \sim n, \quad n^2 - n + 5 \sim n^2, \quad 3n\log n + 100n \sim 3n\log n,$$

but

$$2n \nsim n, \quad n^2 - n + 5 \nsim n\,.$$

The notation $f \sim$ is read as "eff is asymptotically equal to gee" (»asymptotisk lig med«), or just "eff is tilde gee" among friends.

Proposition 1.1 (as a relation). *$\sim$ is an equivalence relation on the set of real functions. That is, it is*

symmetric: *$f \sim g$ if and only if $g \sim f$*
reflexive: *$f \sim f$*
transitive: *if $f \sim g$ and $g \sim h$ then $f \sim h$*

Proposition 1.2 (Rules). *Tilde notation satisfies the following rules:*

1. *If $f_1 \sim g_1$ and $f_2 \sim g_2$ then $f_1 + f_2 \sim g_1 + g_2$.*
2. *If $f_1 \sim g_1$ and $f_2 \sim g_2$ then $f_1 \cdot f_2 \sim g_1 \cdot g_2$.*
3. *$n + C \sim n$ for every constant C*
4. *$n + \log n \sim n$*
5. *$a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0 \sim a_k n^k$*
6. *$\lfloor x \rfloor \sim \lceil x \rceil \sim x$*
7. *$Cn \nsim n$ unless $C = 1$.*

In summary, think of $\sim$ as some weak equality symbol where we disregard lower order terms (but do pay attention to the constant in front of the highest-order term.)

1.1.2 *Asymptotic bound: big Oh.*

The notation $f(x)$ is $O(g(x))$, sometimes abbreviated to $f(x) = O(g(x))$ or $f = O(g)$, "fuzzy version of $f \leq g$." Informally, $f(x) = O(g(x))$ if $f(x)/g(x)$ is not ∞ as x grows. Some usage examples are:

$$n + 1 = O(n), \quad n^2 - n + 5 = O(n^2), \quad 3n\log n + 100n = O(n\log n),$$

and also

$$2n = O(n)\,,$$

but

$$n^2 - n + 5 \text{ is not } O(n)\,.$$

The notation $f = O(g)$ is read as "eff is big-oh of gee", or just "eff is oh of gee", or "eff is order of gee", almost never as "eff equals big-oh of gee". [2]

Proposition 1.3 (as a relation). *The relation between f and g when $f = O(g)$ is*

reflexive: *$f = O(f)$*
transitive: *if $f = O(g)$ and $g = O(h)$ then $f = O(h)$.*

O-notation is *not* symmetric in any sense. First, n^2 is not $O(n)$ (even though $n = O(n^2)$.) Secondly, the combination of letters "$O(n)$ is n^2" makes no sense; O is only defined when it appears on the right side of an equality symbol.

Proposition 1.4 (rules). *O-notation satisfies the following rules:*

1. *if $f \leq g$ then $f = O(g)$ provided f is positive.*
2. *If $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then $f_1 + f_2 = O(g_1 + g_2)$.*
3. *If $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then $f_1 + f_2 = O(\max(g_1, g_2))$.*
4. *If $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then $f_1 \cdot f_2 = O(g_1 \cdot g_2)$.*
5. *If $f = O(g)$ then $Cf = O(g)$ for every constant C.*
6. *$n + \log n = O(n)$*
7. *$n^a = O(n^b)$ for $a \leq b$*
8. *$a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0 = O(n^k)$*
9. *If $f \sim g$ then $f = O(g)$ but not vice versa.*

[1]The term "asymptotic" is highly misleading, it means "not falling together", highlighting the fact that the function $x \mapsto 1/x$ never touches the x-axis (its "asymptote"). This aspect of *not touching* plays no role for us at all, it is in fact *false*: Our asymptotic notions are reflexive, so a function is asymptotically itself. You are advised to ignore your knowledge of Greek and read the word "asymptotically" as a synonym of "for large n".

[2]There are different schools of thought about writing $f = O(g)$, $f(x) = O(g(x))$, $f(x)$ is $O(g(x))$, or even $f \in O(g)$. They all mean the same and have their notational or logical benefits and disadvantages.

Author's address: Thore Husfeldt.

The curious expression $O(1)$ should be read as "any constant". Expressions like $O(3n)$ makes sense, but are confusing and unprofessional and should be avoided; the expression $O(3n)$ means the same as $O(n)$.

In summary, think of O as some weak less-than-or-equal-to symbol where we disregard lower order terms as well as the constant in front of the highest-order term.

1.2 Tilde Explained

1.2.1 *Limits.* Definition, multiplicative, additive, infinity

1.2.2 Tilde. **Definition**. For functions, $f, g\colon \mathbf{R} \rightarrow \mathbf{R}$ we say that f is *asymptotically equal* to g, in symbols

$$f(x) \sim g(x)$$

if

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1\,.$$

Formality. The above makes no sense whenever $g(x)$ is zero infinitely often as x tends to infinity. An example of such a function is $\sin(x)$; it vanishes whenever x is a multiple of π. Luckily for us, such functions do not appear in the analysis of algorithms (running times are not 0), and we could proceed by just ignoring this issue. Formalists will want to read every occurrence of "function" as a "function that is eventually nonzero," i.e., "function f for which there exists a value x_0 such that $f(x) \neq 0$ for $x \geq x_0$." Such formalists will want to convince themselves that all functions appearing the analysis of algorithms (polynomials, logarithms, exponentials and their compositions) satisfy this requirement, except for cases such as $n - n$. Developing this carefully is neither hard nor illuminating.

1.2.3 Properties.

Proposition 1.5. *$\sim$ is an equivalence relation on the set of real functions. That is,*

(1) *$f(x) \sim g(x)$ iff $g(x) \sim f(x)$ (symmetry),*
(2) *$f(x) \sim f(x)$ (reflexivity),*
(3) *if $f(x) \sim g(x)$ and $g(x) \sim h(x)$ then $f(x) \sim h(x)$ (transitivity).*

Proof. Directly from the definition and rules for limit. For instance,

$$\lim_{x\to\infty}\frac{f(x)}{h(x)} = \lim_{x\to\infty}\frac{f(x)}{h(x)}\frac{g(x)}{g(x)} = \lim_{x\to\infty}\frac{f(x)}{g(x)}\frac{g(x)}{h(x)} = \left(\lim_{x\to\infty}\frac{f(x)}{g(x)}\right)\cdot\left(\lim_{x\to\infty}\frac{g(x)}{h(x)}\right) = 1\cdot 1 = 1\,.$$

establishes transitivity except if you worry about $g(x) = 0$. □

Proposition 1.6 (logarithm)**.** *If $f \sim g$ then $\log f \sim \log g$.*

False. If $f \sim g$ then $2^f \sim 2^g$.

Proposition 1.7 (additive, multiplicative)**.** *If $f \sim g$ and $h \sim k$ then $f + g \sim h + k$.*

False (multiplication by constant). If $f \sim g$ then $cf \sim g$ for any constant c.

1.3 Big Oh Explained

1.3.1 Big Oh Defined. Let f and g be real-valued functions defined on the same set of nonnegative real numbers. Then $f(x) = O(g(x))$ if there exists a positive real number B and a nonnegative real number b such that

$$|f(x)| \leq B|g(x)| \qquad \text{for all } x > b\,.$$

Many very similar definitions exist, they all define the same notion.[3]

1.3.2 Properties. **Proposition (addition).**

Proposition (multiplication).

Proposition (scalar multiple).

Proposition (logarithms).

We want to argue things like $\frac{1}{2}n = O(n)$ and $n^2 = O(n^3)$. This should be intuitively easy, since the cube grows faster than the square – for instance, $10^2 < 10^3$ – but note that for instance $(-2)^2 > (-2)^3$ and even $(\frac{1}{2})^2 > (\frac{1}{2})^3$, so the claim holds only for large enough n; in this case, for $n \geq 1$. Also note that $-n^2 \leq -n$ for positive n but $-n^2$ is not $O(-n)$, so cleary we have to be careful

Proposition 1.8 (Domination)**.** *Assume*

(1) *g is eventually at least f, i.e., if there exists b such that $f(x) \leq g(x)$ for $x \geq b$,*
(2) *f is eventually positive, i.e., there exists b' such that $f(x) \geq 0$ for $x \geq b'$.*

Then $f = O(g)$.

Proof. For $x \geq \max(b, b')$, we have $|f(x)| = f(x) \leq g(x) \leq |g(x)|$. □

Proposition 1.9 (ln versus linear)**.** $\ln n = O(n)$.

Proof. We will use Propostion 1.8. The logarithm is positive for $n > 0$. We want to show $\ln n \leq n$ for large enough n, say $n \geq 1$. But that is just calculus. First, for $n = 1$ we have $\ln n \leq n$. Moreover, the function $n \mapsto n - \ln n$ is nondecreasing for $n \geq 1$ since its derivative $1 - \frac{1}{n} > 0$ is nonnegative. □

Proposition 1.10 (power functions)**.** *Let $a \leq b$. Then $n^a = O(n^b)$.*

Proof. It suffices to show that for $n \geq 0$ we have $n^a \leq n^b$; then we can use Proposition 1.8. We need to show that the difference $n^b - n^a$ is nonnegative. Again, we use elementary calculus. The difference is zero for $n = 1$, and it is nondecreasing since its derivative $b - a \geq 0$ is nonnegative. □

Proposition (polynomials).

We are still unable to establish $\log n = O(\sqrt{n})$. Such expressions appear frequently in the analysis of algorithms, for instance when comparing shellsort $O(n^{3/2})$ to mergesort $O(n \log n)$, or $\sqrt{n}$-trees to

[3]Apart from notational substitions, such as C for B and n_0 for b, the seemingly bigger differences are: (i) requiring $x \geq b$ instead of $x > b$. This makes no difference: just make b slightly larger/smaller. (ii) defining the domain of f and g as the natural numbers (rather than the nonnegative reals). For our functions (polynomials, logarithms, etc.), this makes no difference. (iii) more interestingly, dropping the norms, so that the requirement is $f(x) \leq B \cdot g(x)$. The functions we are interested in (running times of algorithms) are nonnegative, so $|f(x)| = f(x)$ anyway and therefore this detail in the definition again makes no difference. However, some internal calculations become easier when the norms are kept.

balanced binary search trees. Unfortunately, I know no short and easy proof for these relationships.

Proposition 1.11 (ln versus power)**.** *For all exponents $a > 0$,*

$$\ln x = O(x^a)\,.$$

Proof using integrals. We have

$$\frac{1}{y} \leq y \qquad (\text{for any } y \geq 1)\,.$$

Thus, for any $z \geq 1$,

$$\int_1^z \frac{1}{y}\,dy \leq \int_1^z y\,dy\,,$$

so that

$$\ln z \leq \tfrac{1}{2} z^2\,.$$

In particular, if we choose $z = x^{a/2}$, then we have $\ln x^{a/2} \leq \frac{1}{2}(x^{a/2})^2$, which simplifies to

$$\ln x \leq x^a \qquad (x \geq 1)\,.$$

Now we can apply Proposition 1.8 to conclude $\ln n = O(x^a)$. □

Proof using l'Hôpital's rule. We have

$$\lim_{x\to\infty} \frac{\ln x}{x^a} = \lim_{x\to\infty} \frac{1/x}{ax^{a-1}} = \lim_{x\to\infty} \frac{1}{ax^a} = 0\,.$$

In particular, the limit is bounded away from infinity. Conclusion by Proposition (not yet established). □

Proposition 1.12 (polylog versus power)**.** *For all bases $r > 1$ and exponents $a > 0$ and $k > 0$,*

$$(\log_r x)^k = O(x^a)\,.$$

Proof. From proposition L and the previous result we conclude $\log b = O(x^a)$. □

1.3.3 *Big Oh as a limit.*

1.4 Multivariate asymptotics

2 AMORTISATION

Consider the resizing array implementation of `Stack` (Algorithm 1.1) and the collowing claim:

Proposition 2.1 (Proposition E in SW 1.4, only for push)**.** *In the resizing array implementation of* `Stack` *(Algorithm 1.1), the average number of array accesses per operation for any sequence of* `push()` *operations starting from an empty data structure is constant in the worst case.*

This is a true statement, and proved in [SW]. We include a full proof below, in section 3.1, just for completeness.

Exercise 1. Is Prop. 2.1 still true when I remove "starting from an empty data structure"?

Exercise 2. Is Prop. 2.1 still true when I remove "average"?

2.1 Terminology: Amortisation versus average

From now on, we will try to avoid the term "on the average." Not because it is wrong, but because it is too broad. For more precision, we introduce the concept of "amortised cost", borrowing terminology from the world of finance.[4] [5]

The idea is to "write off" the costs for `reduce()` in the long run by showing that, even though a particular call may be costly, these expensive calls happen with such low frequency that by instead charging a slight cost to every operation, the expensive operation will be paid.

As with many good analogies, there is a pitfall: In banking it is perfectly acceptable to expend a large amount of money immediately, and amortise it later, for example, buying a house off a large loan, or amortising the expense of a good-quality tool after many uses. In contrast, this is not acceptable in the analysis of algorithms: We will insist that every expensive operation is already paid (in small rates) beforehand.[6] Maybe "piggy bank" analysis would have been a better term, but the word is what it is.

So, to be quite precise, here is the definition:

Definition 2.1 (Amortized cost)**.** Let $T(N)$ be the worst-case[7] total running time for any sequence of N operations. The *amortized* time for each operation is $T(N)/N$.

So why don't we just call it "average" time?

Because "average" also means many other things. For instance, we could average over operations (say, call `push()` and `pop()` with equal probability, and with random arguments to `push()`)—this is called "average case complexity", and a difficult and active reseach area. Also, for algorithms that use randomness, we could average over tho random choices of the algorithm—this is called "expected running time", another active research area. Both of these concepts of average make sense for a single opeation, and both would require us to be quite precise about the random processes involved.

In contrast, "amortized time" is a particular, precise, worst-case average over a sequence of operations.

Exercise 3. Bob is a reluctant runner. He has been running 4 km every day of the week, except in the week-ends. (i) What is the worst case number of kilometers he runs per day? (ii) He started this new workout schedule on a Saturday. What is the amortised number of kilometers he runs per day?

Exercise 4. Ran is also a reluctant runner. He rolls a die every morning and runs as many kilometers. (i) What is the expected number of kilometers Ran runs per day, assuming he has a perfect die? (ii) What is the worst-case number of kilometers Ran runs per day? (iii) For any sequence of days starting on a Saturday, what is the worst-case amortised number of kilometers Ran runs per day? (Note that this answer must hold *even if his dice are cursed.*)

[4]**amortize** /ˈæmərˌtaɪz, əˈmɔrtaɪz/ verb [trans.] reduce or extinguish an amount by money regularly put aside: *loan fees can be amortized over the life of the mortgage.*, gradually write off the initial cost of an asset: *they want to amortize the tooling costs quickly.*

[5]Many people view this material as difficult, subtle, and somewhat boring. It may be well characterised by a word whose latin root is *ad mortis*, "to death."

[6]Another place where the banking analogy breaks down is that banks charge interest. Algorithms don't.

[7]Imagine that the sequence of operations, including the parameters to these operations, are chosen by your worst enemy.

Exercise 5. My phone company charges 10 DKK for 1 minute of voice call. This is exorbitant, but I never use my phone for making a phone call anyway. According to the contract, I have to pay at least 100 DKK per month for voice calls whether I use them or not, but the contract automatically 'rolls over' the unused minutes to the next month. I have call my mum for Christmas for a 2-hour call. Describe the expense in terms of 'money I spend on voice calls each month' in the worst case and in the amortised sense.

Exercise 6. A multiride ticket in the Danish amusement part Tivoli costs 200 DKK and is valid for 10 rides. What is the worst case cost for a single ride? What is the amortised cost for a ride? (*Careful!*)

3 AGGREGATE METHOD

3.1 Stack without Popping

Proposition 3.1 (Proposition E in [SW 1.4], reformulated)**.** *In the resizing array implementation of* `Stack` *[SW, Algorithm 1.1], the push operation requires amortised constant time.*

Proof. Starting from an empty data structure, consider any sequence of push operations of length k. Our cost model is the number of array accesses. The second line in the body of push always takes exactly one array acccess, for a total of k accesses.

The difficult part are the array accesses resulting from the calls to `resize`. A call to `resize`(m) when the data structure contains N elements requires $m + 2N$ array accesses. When `resize` is called from push, we have $m = 2N$, so $m + 2N = 4N$. Such a call happens when N is a power of 2, i.e., for $N = 1, 2, 4, 8, \cdots, k'$, where k' is the largest power of 2 such that $k' \leq k$. In summary, the array accesses resulting from calls to resize result in at most

$$4(1 + 2 + 4 + 8 + \cdots + k') = 4(2k' - 1) \leq 8k$$

array accesses. We conclude that the total number of array accesses for any sequence of k calls to push is $9k$. □

3.2 Mechanical Counter

Whenever a mechanical counter is incremented, the rightmost wheel (the "ones") is turned. Every 10th increment, when the rightmost wheel transitions from 9 to 0, its left neighbour is turned as well. This effect continues to the left, so after $10^k - 1$ increments, when the counter shows k many 9s, the next increment will lead $k + 1$ wheels to turn.

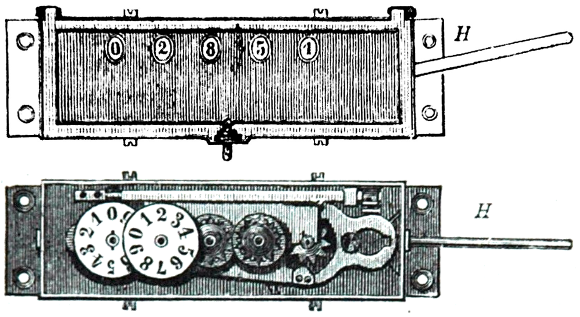

Fig. 1. A 5-digit mechanical counter.

In our terminology, the worst case number of operations for N increments is logarithmic. But in the amortised sense, the behaviour is much better:

Proposition 3.2. *Starting from 0, a mechanical counter needs a constant amortised number of operations per increment.*

Proof. Consider a sequence of $N \geq 0$ increments. Some of them will turn exatly 1 wheel (namely the rightmost), others will turn more wheels. The trick is and to look at each wheel separately. Enumerate the wheels $0, \ldots, k-1$ from right to left, so wheel 0 shows the least significant digits. Note that the number of wheels needed is exactly the number of digits of N (in the decimal representation), which is $k = \lfloor \log_{10} N \rfloor + 1$.

Wheel 0 was rotated N times. Wheel 1 was rotated $\lfloor N/10 \rfloor$ times. In general, wheel r was turned

$$\left\lfloor \frac{N}{10^r} \right\rfloor \quad \text{times} \qquad (0 \leq r < k)\,.$$

Adding these contributions for the total number of operations, gives

$$\left\lfloor \frac{N}{10^0} \right\rfloor + \left\lfloor \frac{N}{10^1} \right\rfloor + \cdots + \left\lfloor \frac{N}{10^{k-1}} \right\rfloor \leq$$
$$\frac{N}{10^0} + \frac{N}{10^1} + \cdots + \frac{N}{10^{k-1}} = N\left(\frac{1}{10^0} + \frac{1}{10^1} + \cdots + \frac{1}{10^{k-1}}\right) =$$
$$N\left(1 + \frac{1}{10} + \frac{1}{100} + \cdots + \frac{1}{10^{k-1}}\right) < 2N.$$

This finishes the proof. □

Exercise 7. Repeat the above argument, but for a *binary* counter. (Every wheel contains only two digits, 0 and 1.)

4 ACCOUNTING METHOD

What we will do now is to show that Prop. 3.1 is still true when I remove "`push()`":

Proposition 4.1 (Proposition E in SW 1.4 in full, reformulated)**.** *In the resizing array implementation of* `Stack` *(SW Algorithm 1.1), the amortized number of array accesses per operation is constant in the worst case.*

So now the claim is stronger; it talks about any sequence of operations (including pops), instead of only pushes. Intuitively, the claim sounds reasonable, since `pop()` only makes the stack smaller, whereas the really expensive operation happens only when the stack grows and causes and expensive `resize()`. However, even though this argument sounds reasonable, it is *wrong*. The next section shows why.

4.1 Stingy resizing array

We will change algorithm 1.1 to halve the size of `a` as soon as it can. More specifically, let algorithm 1.1' be the same as algorithm 1.1, with the following change in the implementation of `pop()`, where a 4 was replaced by a 2. To be quite precise, we change the line

```
    if (N > 0 && N == a.length/4) resize(a.length/2);
```

to

```
    if (N > 0 && N == a.length/2) resize(a.length/2);
```

Note that the implementation remains *correct* (in the sense that it correctly implements a stack), and it will even save space. However, we can no longer guarantee the performance of Prop. 4.1.

Exercise 8. Exhibit a sequence of operations for which algorithm 1.1' requires a quadratic number of array accesses.

In other words, the sequence of operations from the preceding exercise requiers a linear number of array accesses *per operation* on the average, much worse than the constant number promised by Prop. 4.1. We must resign ourselves to the fact that Prop. 4.1 does not hold for algorithm 1.1'. Since it does hold for algorithm 1.1, the argument needs to become quantitative somewhere—it must acknowledge the difference between 4 and 2. Our intuitive argument was purely qualitative, so it can't work.

4.2 Amortized Analysis of Resizing Array

We rephrase the claim with our new terminology:

Proposition 4.2 (Prop 1.1, rephrased)**.** *In the resizing array implementation of* `Stack` *(Algorithm 1.1), the amortised number of array accesses per operation for any sequence of operations starting from an empty data structure is constant in the worst case.*

Proof. We will pretend that every array access costs one coin. We associate a (ficticious) piggy bank with our data structure and make every call of `push()` and `pop()` deposit a certain number of coins in the bank. (Determining how many coins is the technically difficult part of the argument.) The aim is to have the pig bank-roll every call of `resize()`.

Let my pull the right constants out of my hat: `push()` shall deposit 8 coins, `pop()` shall deposit 4 coin.[8]

The difficulty are the two internal calls to `resize()`. A call to `resize(max)` requires `max` + $2N$ array accesses.[9] We can simplify this expression by observing that `max` = $2N$ whenever `resize()` is called. To see this, look at the condition to `if` in both calls to `resize()`: from inside `push()`, we have `max` = 2 ∗ `a.length` = $2N$, and from inside `pop()` we have `max` = `a.length/2` = `2a.length/4` = $2N$. Thus, both calls of `resize()` require $4N$ coins. We need to show that the pig can handle that.

We need to consider the very first call of `resize()` separately. This is an easy case. The data structure is initialised with `a.length` = 1 and $N = 0$, and the first call of `resize()` must come from `push()` when $N = 1$. We need $4N = 4$ coins, and have just deposited 8 coins, so the charge is easily paid.

Another easy observation will turn out to be useful, so I will pull that out of my hat as well: Immediately after every call of `resize()`, there are exactly N occupied and N free cells in `a`: from looking at `resize()` we can see is that there are `max` − N free cells, and we already observed above that `max` = $2N$.

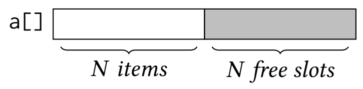

Fig. 2. The data structure immediately after `resize()`.

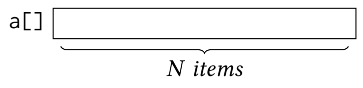

Fig. 3. The data structure immediately before `push()` calls `resize()`.

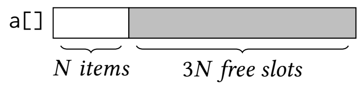

Fig. 4. The data structure immediately before `pop()` calls `resize()`.

With this observation, we can proceed to show that at the start of every subsequent call of `resize()`, there are at least $4N$ coins in the bank.

- If the call came from `push()`, i.e., to prevent overflow, then there must have been at least $N/2$ calls to `push()` since the last `resize()`. Each `push()` deposited 8 coins, so there are $4N$ coins, as needed.
- If the call came from `pop()`, i.e., to prevent underflow, then there must have been at least N calls to `pop()` since the last `resize()`. Each `pop()` deposited 4 coins, so there are $4N$ coins, as needed.

□

Exercise 9. Can we charge all costs to `push()`? How much does it have to pay if `pop()` pays nothing?

Exercise 10. Can we charge all costs to `pop()`? How much does it have to pay if `push()` pays nothing?

4.3 Localising the piggy bank argument

Sometimes it is easier to localise the accounting argument by removing the fictional piggy bank and depositing the coins in the data structure instead.

For example, we can say that push operation puts 8 coins in the cell it just filled, and every pop puts 4 coins in the cell that it just freed.

As before, when doubling the array from N to $2N$, we argue that at least N `push()` operations happened, so the entries in `a` from $N/2 - 1$ to N contain at east 8 coins each, for a total of $4N$ coins. Similarly, for halving the array from $4N$ to $2N$, the calls to `pop()` put 4 coins on each of the free cells in $N + 1, \ldots, 2N$, for a total of $4N$ coins.

The local argument is sometimes attractive because there is a strong inuition about what the coins will be doing, should they ever be needed. For example, the 4 coins deposited by `pop` in position `a[N+1]` will be able to pay for the allocation of the new array cells `temp[1]` and `temp[N+1]`, and the two accesses in `temp[1] = a[1]`.

[8]If you want to be precise `push()` costs 9 coins. It uses 1 coin to pay for the array access in line 2, and puts the others in the pig. Similarly, `pop()` costs 6 coins. It uses 2 coins to pay for the array acccesses in its first two lines and puts the remainder in the pig.
[9]See the last Q&A in section 1.4 for our convention about the cost of new.

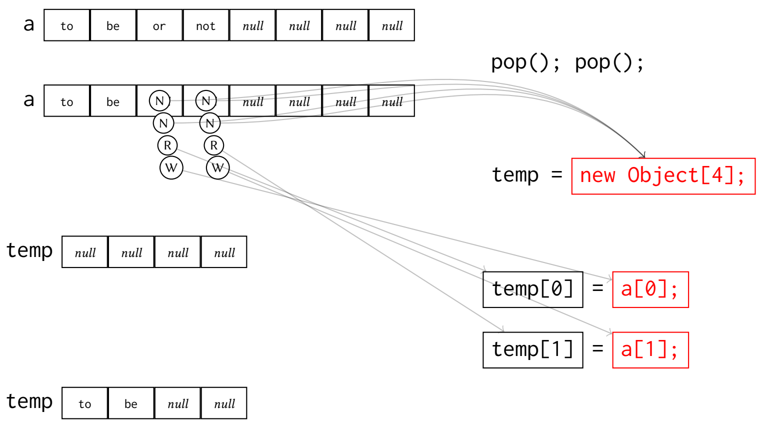

Fig. 5. Two pop() operations each deposit 4 coins (marked N, N, R, W), in a[2] and a[3] respectively. When resize() is called, the coins marked N are spent allocated the new array temp, the R coin is spent reading from a, and the W coin is spent writing to temp. In the new a (which is temp), there is no money.

But it's a matter of taste. I like the local argument better for exercise 9.

Exercise 11. Prove proposition C using the piggy bank method instead. If you use the local argument, where does the money go? How little do you actually need? [10]

Exercise 12. Assume the counter is built out of increasingly heavy material. Each wheel costs twice as much to turn as the previous. (So, turning wheel 0 costs 1, wheel 1 costs 2, wheel 2 costs 4, . . ., wheel k costs 2^k.) Does the analysis still work? What if wheel k costs 10^k? 11^k?

5 UNION–FIND VARIANTS

This makes sense after [SW] 1.5

In this section we consider two variants of union–find, and present careful analyses of their amortised running times.

These serve primarily as examples of nontrivial applications of the aggregate method; the weighted quick-union structure is better than both of our variants in this section. However, both data structures are useful examples of more general algorithmic design principles that are useful in other contexts:

(1) if you have to update parts of a data structure, favour decision that update the *smaller* part. This is the idea behind the *weighted* version of quick-find.
(2) *memoization*: if you've performed a computation, store its result, so as to avoid recomputing it later.

Both of these ideas are intuitively clever, and worst-case analysis is unable to establish this intuition. In contrast, the amortized analysis shows in both variants that they are worth doing.

5.1 Weighted quick-find

We return to the first, very simple union-find data structure of [SW], called *quick-find*.

Let us try to salvage this idea, at least partially. We're not going to be able to beat the quick-union implementations, but this section

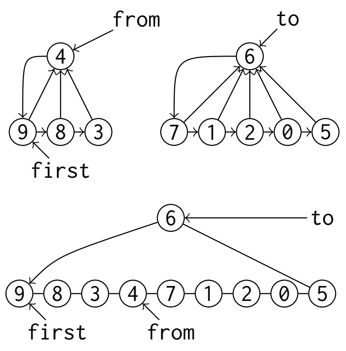

Fig. 6. union(1,8), where id[8] == 4 and id[1] == 6. The next pointers are shown.

is mainly about *analysis*, not so much about presenting the best union–find algorithm known to Man.

Our first improvement is to make a clever choice about whether union(p, q) renames p's component to q's or vice versa. Our choice is to always rename the smaller component. For this, we introduce another array int[] sz to store component sizes, initially all 1, and with the same meaning as in algorithm 1.4.

The second improvement is to link all elements of a component, so that we can quickly iterate over them when we want to rename them, instead of iterating over all of id. For this, we implement yet another array int[] next of indices, so that next[i] is the element following i in some circular list of the component ids.

For example, the data structure looks like this after the operations described by tinyUF.txt.:

	0	1	2	3	4	5	6	7	8	9
id[]	6	6	6	4	4	6	6	6	4	4
next[]	5	2	0	4	9	6	7	1	3	8
sz[]	1	1	3	1	4	1	6	1	1	1

Figure 6 shows the this situation, and the situation after another call to union(). An implementation is given in figure 7.

[10]To make this entertaining, assume it takes one pound sterling to turn a wheel and give the answer in pence. Next, assume it takes 1 shilling.

```
public class WeightedQuickFind {
  private int[] id, sz, next;
  private int count;

  public WeightedQuickFind(int N) {
    count = N;
    id = new int[N];
    next = new int[N];
    sz = new int[N];
    for (int i = 0; i < N; i++) {
      id[i] = i;
      next[i] = i;
      sz[i] = 1;
    }
  }

  private void rename(int from, int to) {
    int first = next[from];
    for (int i = first; i != from; i = next[i])
      id[i] = to;
    id[from] = to;
    next[from] = next[to];
    next[to] = first;
    sz[to] += sz[from];
  }

  public void union(int p, int q) {
    int pID = find(p);
    int qID = find(q);
    if (pID == qID) return;
    if (sz[pID] < sz[qID]) rename(pID,qID);
    else                   rename(qID,pID);
    count -= 1;
  }
}
```

Fig. 7. Weighted quick-find. See [SW, sec. 1.5] for find.

Intuitively, we have made our data structure a lot faster. For instance, consider the sequence of $N-2$ unions with arguments $(0,1), (1,2), (2,3), \ldots, (N-2, N-1)$. Each operations takes constant time, whereas the original `QuickFind` data structure would have used linear time. This is great progress. Unfortunately, our usual way of analyses is unable to detect that improvement, because we usually are interested in *worst-case* running times. And the worst case running time of `WeightedQuickFind` is quite bad:

Exercise 13. Exhibit a call to `union()` for which `WeightedQuickFind` that takes linear time.

Again, the proper way to analyse the data structure is the amortized point of view. [11]

Proposition 5.1. *In weighted quick-find, the* `union()` *operation takes amortized logarithmic time in the worst case, beginning from an initialized data structure.*

Proof. We can show this using the aggregate method, so we consider a sequence of k calls to `union()` and want to show that it takes $O(k \log k)$ time.

We will count the number of updates of `id[]` in `rename()`, from the perspective of a single site i. The crucial observation is that whenever the `id` of i is renamed, the size of its component at least doubles. (This is because we always rename the smaller of two unioned components, and the size of the union is at least twice of the smaller one.) On the other hand, no component can ever be larger than $k+1$. (This is because every elemented started out in a component of its own. At some time, that component must have been united with the others, which must have required a call to `union()`.) Thus, the component of i has been doubled no more than $\log(k+1)$ times. In particular, `id[i]` has been changed at most $\log(k+1)$ times.

Next, we need to argue that at most $2k$ sites were renamed during the k operations. This is because a site that never appeared as one of the two arguments in some call to `union` still remains in its original component. Thus, at most $2k$ sites are ever renamed. We argued above that just saw that each site gets renamed at most $\sim \log k$ times in total. Thus, the total number of updates to `id` is $\sim k \log k$. This finishes the proof of propostion A4. □

Let us reiterate a few points about the formulation of Prop. 5.1

(1) Note the role that 'worst case' plays: It says that in *any*, even the most maliciously constructed input sequence, is is *impossible* that the first k operations take more than $O(k \log k)$ time in total. (Without 'worst case', proposition A4 remains true, since it also holds for 'best case' or 'average case' or 'typical case', however you want to define them. But none of those statements are interesting, and all of them are weaker.)

[11]Before we continue, let's remind ourselves that there is a good reason for our usual focus on the worst case. We normally don't evaluate our data structure by its behaviour on particularly well-chosen inputs that make the data structure look good. The *best-case* time for `QuickFind` was constant as well, after all: Just call `union(0,1)` N times in a row.

Also note that 'typical case' (however that should be defined) is a no-starter. Almost all calls to union in our two examples take constant time, so the 'typical case behaviour is constant for WeightedQuickFind'—but that says more about our input sequences than the data structure.

At this time one is tempted to define 'average case,' or, to be quite precise 'average input sequence.' This can indeed be done quite rigorously, by defining a random process. For instance, we could pick $p, q \in \{0, \ldots, N-1\}$ at random, independently and with the same probability, and then try to determine the expected value of the running time of `union(`p,q`)` after N operations. There are two reasons for us to *not* pursue this method of analysis. First, it is mathematically sophisticated and requires some exposure to discrete probability theory. Second, it tells us surprisingly *little* about our data structure, because the result depends very heavily on the choice of distribution. (For instance, why should p and q be independent? Why should they be uniformly distributed instead of normal, or Poisson, or a dozen other distributions that appear in real life?) To appreciate these questions, imagine the union–find data structure used in an algorithm for Kruskal's algorithm for minimum spanning tree. Then the p and q depend on the structure of the input graph and the weight of its edges—that is absolutely not a uniformly random process! It would be much more interesting to define, say 'the input sequence arising from the union–find calls of an execution of Kruskal's algorithm on a random graph.' But this gets *very* difficult to analyse (including specific details about the implementation of Kruskal!), and only shifts the question to 'what kind or random process did the graph arise from'? No matter how we try to define 'random inputs,' we find ourselves in the position of having to motivate the distribution.

The discussion above largely served to motivate why we try to avoid the term 'average.' It's not very well defined, and of very questionable value. (Moreover, we can't do the math.)

(2) Note the role that ‘amortized’ plays: Without it, proposition 4A is plainly false; as shown by Exercise 13.
(3) Note the role that ‘beginning from an initialized data structure’ plays. Without it, proposition 4A is plainly false: if we begin later, Exercise 13 shows that there are sequences of length 1 (namely “union(0,1)”) that take time linear in the size of the data structure. And if we begin earlier, then the initialisation itself takes linear time.

5.2 Unweighted Quick-Union with Path Compression

Assume that we extend the (unweighted) quick-union algorithm (SW, pp. 224) to include *path compression* (but not union-by-rank). To be precise, we rewrite the `find(int p)`-operation so that it links every site on the path from p to the root directly to the root, like in Fig. 8

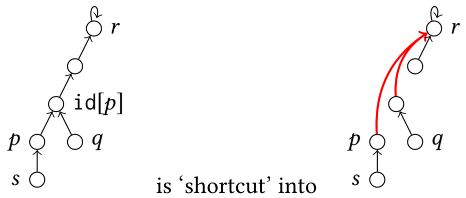

Fig. 8. Path compression resulting from the call `find(p)`. The edges on the path from p to the root r, except for the last edge, are removed and replaced by new edges (shown red.) The resulting distance to the root for every node on the path is now 1.

The idea is that since the algorithm is ‘touching’ each of these nodes anyway, it might as well spend slightly more time shortcutting them to link directly to the root, so that *next* time they’ll be close to it.

One elegant way of implementing that is recursively, as follows:

```
public int find(int p) {
    if (id[p] == p) return p;
    int r = find(id[p]);
    id[p] = r;
    return r;
}
```

Exercise 14. Give a nonrecursive implementation of the same idea.

We will show that this data structure is as good as *weighted* quick-union, at least in the amortised sense:

Proposition 5.2 (Exercise 1.5.12 in SW)**.** *(Unweighted) quick-union with path compression requires amortised logarithmic time per update.*

Before proving this, convince yourself that amortisation is crucial for this claim. For unlike for weighted quick-union, we cannot establish a logarithmic time bound *in the worst case*:

Exercise 15. Exhibit a sequence of operations that shows that (unweighted) quick-union with path compression requires linear time per update in the worst case.

Proof of Prop. 5.2. We need to consider an arbitrary sequence of N operations (unions or finds) and show that the total time for these operations is $O(N \log N)$. Our cost model will be the total number of edges that exist at any time during the evolution of the data structure.

We let T_u denote the subtree rooted at node u, including u itself. An edge e from u to v is called *heavy* if, at the time of its destruction, it accounted for at least half of its parent’s weight in the sense that $|T_u| \geq \frac{1}{2}|T_v|$. (A self-loop is always heavy.) Otherwise it is *light*; this includes those edges that are never destroyed. We show that both the number of light edges and the number of heavy edges are at most $O(n \log n)$.

Light: There is a negligible number of n (light) edges that are never destroyed and survive to the end. For the others, a light edge is destroyed only when it lies on the path traversed by a find operation. Consider such a path $v_0, \ldots, v_r$ from top to bottom, and let e from u to v be one of its edges. Because e is light, the corresponding subtrees double in size: $|T_v| > 2|T_u|$. Thus, if the path contains k light edges, we have

$$|T_{v_r}| > 2^k |T_{v_0}|\,.$$

Since $|T_{v_r}| \leq n$ and $|T_{v_0}| \geq 1$, we have $k \leq \log_2 n$. Thus, the path contains at most $\log_2 n$ light edges, and therefore the find operation can not destroy more than that. Since there are at most $3n$ find operations, we conclude that the total number of light edges is $O(n \log n)$.

Heavy: We associate each heavy edge e from u to v with its target v. Let $e_0, \ldots, e_r$ be the heavy edges pointing to v at any time during the evolution of the data structure. In particular, e_0 is the initial self-loop from v to v. Let $w_0, \ldots, w_r$ denote the *weight* of v, i.e., the size $|T_v|$ of the subtree rooted at v, at the time when $e_0, \ldots, e_r$ are destroyed. After e_0 is destroyed (by a union operation) and vertex v ceases to be a root, its weight can only decrease. Moreover, because each e_i is heavy, we know that the removals of $e_1, \ldots, e_r$ each reduced the weight of v by at least half. Thus,

$$w_0 \geq w_1 \geq 2w_2 \geq 2^2 w_3 \cdots \geq 2^{r-1} w_r\,.$$

Since each w_i is an integer between 1 and n, we have $r - 1 \leq \log_2 n$. There are n different target vertices, so the total number of heavy edges is $O(n \log_2 n)$. □

Exercise 16. The idea of *path halving* is a simpler version of path compression. The find operation merely connects every vertex to its granparent on the path, instead of to the root. Does the above proof still apply?

6 ON THE WORD ‘AVERAGE’ IN [SW]

We already discussed the use of the word ‘average’ and its various meanings. The concept described in these notes is a specific kind of average called ‘amortised,’ and we have tried to be disciplined in our use of it. This does not mean that the mental model of ‘amortisation is just some kind of average’ is false. In particular, the textbook [SW] itself uses the word in its own formulation of proposition A4.

However, as a possible source of confusion the word ‘average’ is used in Proposition K in [SW, Sec. 2.3] with a different meaning:

Proposition K. Quicksort uses $\sim 2N \ln N$ compares (and one-sixth that many exchanges) on the average to sort an array of length N with distinct keys.

This statement describes the running time of a single run of an algorithm on a random input (namely, a sequence of uniformly and independently distributed integers). This is precisely *not* how 'average' is used in proposition A4: There, the sequence (which defines repeated operations on a data structure) is nonrandom (it is, in fact, a worst case sequence.)

7 ANSWERS

1. No. To see this, begin with a stack that is the result of $2^k - 1$ applications of `push(null)`. Now `N` equals $2^k - 1$ and `a.length` equals 2^k. From this starting position, a single `push()` will result in a call to `resize()`, leading to a number of array accesses that is linear in `N`.

2. No.

3. (i) 4 km per day. (ii) When we measure on the wth Friday (the worst possible day) we get $(0 + 0 + 4 + 4 + 4 + 4 + 4)w$ kilometers in total, for $7w$ days, which means $\frac{20}{7} < 2.9$ kilometers amortised.

4. (i) $\frac{7}{2}$. (ii) 6. (iii) $\frac{5\cdot 6}{7} = \frac{30}{7} \sim 4.28$.

5. In the worst month (December), I spend 1200 DKK (which are indeed removed from my account of unused talk minutes at the phone company). In the amortized sense, I spend 100 DKK per month (which are indeed removed from my bank account). If you want to be precise, the contract must have started in January.

7. Now $k = \lceil \log_2 N \rceil + 1$. The total number of rotations becomes $\lceil N/2^0 \rceil + \cdots + \lceil N/2^{k-1} \rceil \leq N/(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots) = 2N$.

13. Begin with $k-2$ calls to union with arguments $(0, 2), (1, 3), (2, 4), \ldots, (k - 2, k)$. At this stage, all the odd elements are in their own component, and all the even elements are in their own component, each of size $\sim \frac{1}{2}k$. Now a single call to union(0, 1) will have to access $\sim 12k$ array elements of `id`.

15. Begin with N unions `union(0,1)`, `union(1,2)`, . . ., `union(`$N-$`2,`$N-1$`)`. (Each took constant time, the data structure is now a directed path of length N.) Now call `find(0)`. This single operation requires linear time, so the worst case time for an operation is (at least) linear.

16. Almost. The cost model is still valid, since number of edges still accounts for the number of operations, up to linear terms. But it is less clear that a vertex only ever loses weight after it ceases to be a vertex. (After all, path halving may connect a vertex to a fresh parent that is not a root.) However, no vertex ever receices a new *descendant* after it loses its self-loop, so the argument remains true.