

How Airplanes Fly



HACKERMONTHLY

Issue 21 February 2012

Curator

Lim Cheng Soon

Contributors

David Anderson
Scott Eberhardt
Peiter Buick
Pat Shaughnessy
Nick Johnson
Ian Ward
Brandon Minter
Timothy Daly
James Tauber
Lisa Zhang

Proofreaders

Emily Griffin
Sigmarie Soto

Printer

MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be “anything that gratifies one’s intellectual curiosity.” Every month, we select from the top voted articles on Hacker News and print them in magazine format. For more, visit *hackermonthly.com*.

Advertising

ads@hackermonthly.com

Contact

contact@hackermonthly.com

Published by

Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.



Cover Photography: Dave Morrow's Custom Creations

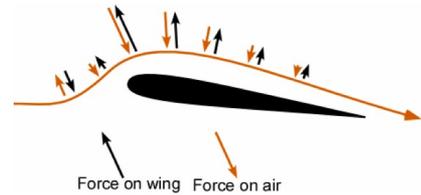
Hacker Monthly is published by Netizens Media and not affiliated with Y Combinator in any way.

Contents

FEATURES

04 How Airplanes Fly

By DAVID ANDERSON *and* SCOTT EBERHARDT



DESIGN

14 The Messy Art of UX Sketching

By PEITER BUICK

PROGRAMMING

20 Never Create Ruby Strings Longer Than 23 Characters

By PAT SHAUGHNESSY

24 Fountain Codes

By NICK JOHNSON

27 Unfortunate Python

By IAN WARD

30 Tips for Remote Unix Work

By BRANDON MINTERN

34 Being a Great Coder

By TIMOTHY DALY

SPECIAL

36 Why 13th Chords

By JAMES TAUBER

38 Elevator Algorithms

By LISA ZHANG



How Airplanes Fly

A Physical Description of Lift

By DAVID ANDERSON & SCOTT EBERHARDT



Photo: Black & White Bi-Plane, [flickr.com/photos/daves-f-stop/5516483143](https://www.flickr.com/photos/daves-f-stop/5516483143)

ALMOST EVERYONE TODAY has flown in an airplane. Many ask the simple question “what makes an airplane fly?” The answer one frequently gets is misleading and often just plain wrong. As an example, most descriptions of the physics of lift fixate on the shape of the wing (i.e. airfoil) as the key factor in understanding lift. The wings in these descriptions have a bulge on the top so that the air must travel farther over the top than under the wing. Yet we all know that wings fly quite well upside down where the shape of the wing is inverted. To

cover for this paradox we sometimes see a description for inverted flight that is different than for normal flight. In reality the shape of the wing has little to do with how lift is generated and everything to do with efficiency in cruise and stall characteristics. Any description that relies on the shape of the wing is wrong.

Let us look at two examples of successful wings that clearly violate the descriptions that rely on the shape of the wing. The first example is a very old design. Figure 1 shows a photograph of the Curtiss 1911 model D type IV pusher. Clearly the air travels the

same distance over the top and the bottom of the wing. Yet this airplane flew and was the second airplane purchased by the US Army in 1911.

The second example of a wing that violates the idea that lift is dependent on the shape of the wing is of a very modern wing. Figure 2 shows the profile of the Whitcomb Supercritical Airfoil (NASA/Langley SC(2)-0714). This wing is basically flat on top with the curvature on the bottom. Though its shape may seem contrary to the popular view of the shape of wings, this airfoil is the foundation of the wings modern airliners.



Figure 1. Curtiss 1911 model D type IV pusher



Figure 2. Whitcomb Supercritical Airfoil

The emphasis on the wing shape in many explanations of lift is based on the Principle of Equal Transit Times. This assertion mistakenly states the air going around a wing must take the same length of time, whether going over or under, to get to the trailing edge. The argument goes that since the air goes farther over the top of the wing it has to go faster, and with Bernoulli's principle we have lift. Knowing that equal transit times is not defensible the statement is often softened to say that since the air going over the top must go farther it must be faster. But, this is again just a variation on the idea of equal transit times. In reality, equal transit times holds only for a wing without lift. Figure 3 shows a simulation of the airflow around a wing with lift.

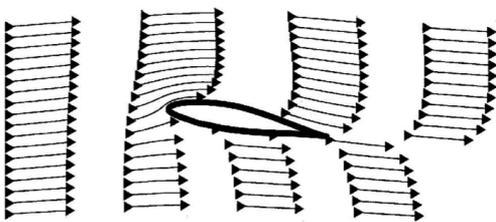


Figure 3 Air over a wing with lift.

The Bernoulli equation is a statement of the conservation of energy. It is correct, but not applicable to the description of lift on a real wing. The wings of an 800,000 pound airplane are doing a great deal of work to keep the airplane in the air. They are

adding a large amount of energy to the air. One of the requirements of the application of the Bernoulli principle is that no energy is added to the system. Thus, the speed and pressure of the air above a real wing in flight are not related by the Bernoulli principle. Also, descriptions of lift that evoke the Bernoulli principle depend on the shape of the wing. As already stated, the shape of the wing affects the efficiency and stall characteristics of the wing but not the lift. That is left to the angle of attack and speed.

Newton's laws and lift

So, how does a wing generate lift? To begin to understand lift we must review Newton's first and third laws. (We will introduce Newton's second law a little later.) Newton's first law states:

A body at rest will remain at rest, or a body in motion will continue in straight-line motion unless subjected to an external applied force.

That means, if one sees a bend in the flow of air, or if air originally at rest is accelerated into motion, a force is acting on it.

Newton's third law states that:

For every action there is an equal and opposite reaction.

As an example, an object sitting on a table exerts a force on the table (its weight) and the table puts an equal and opposite force on the object to hold it up. In order to generate lift a wing must do something to the air. What the wing does to the air is the action while lift is the reaction.

Let's compare two figures used to show streamlines over a wing. In figure 4 the air comes straight at the wing, bends around it, and then leaves straight behind the wing. We have all seen similar pictures, even in flight manuals. But, the air leaves the wing exactly as it appeared ahead of the wing. There is no net action on the air so there can be no lift! Figure 5 shows the streamlines, as they should be drawn. The air passes over the wing and is bent down. Newton's first law says that there must be a force on the air to bend it down (the action). Newton's third law says that there must be an equal and opposite force (up) on the wing (the reaction). To generate lift a wing must divert lots of air down.



Figure 4. Common depiction of airflow over a wing. This wing has no lift.



Figure 5. True airflow over a wing with lift showing upwash and downwash.

The lift of a wing is equal to the change in momentum of the air it is diverting down. Momentum is the product of mass and velocity (mv). The most common form of Newton's second law is $F=ma$, or force equal mass times acceleration. The law in this form gives the force necessary to accelerate an object of a certain mass. An alternate form of Newton's second law can be written:

The lift of a wing is proportional to the amount of air diverted down times the vertical velocity of that air.

It is that simple. For more lift the wing can either divert more air (mass), increase its vertical velocity or a combination of the two. This vertical velocity behind the wing is the vertical component of the "downwash." Figure 6 shows how the downwash appears to the pilot (or in a wind tunnel). The figure also shows how the downwash appears to an observer on the ground watching the wing go by. To the pilot the air is coming off the wing at roughly the angle of attack and at about the speed of the airplane. To the observer on the ground, if he or she could see the air, it would be coming off the wing almost vertically at a relatively slow speed. The greater the angle of attack of the wing the greater the vertical velocity of the air. Likewise, for a given angle of attack, the greater the speed of the wing the greater the vertical velocity of the air. Both the increase in the speed and the increase of the angle of attack increase the length of the vertical velocity arrow. It is this vertical velocity that gives the wing lift.

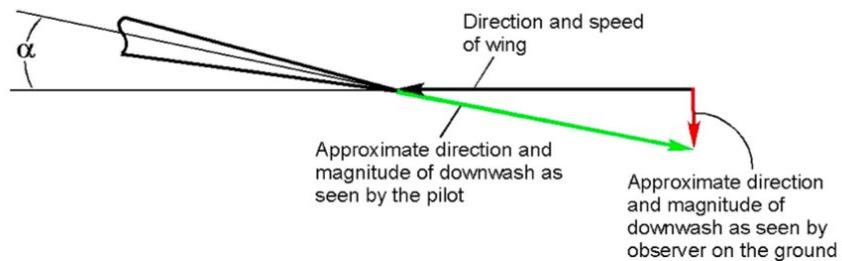


Figure 6. How downwash appears to a pilot and to an observer on the ground.

As stated, an observer on the ground would see the air going almost straight down behind the plane. This can be demonstrated by observing the tight column of air behind a propeller, a household fan, or under the rotors of a helicopter; all of which are rotating wings. If the air were coming off the blades at an angle the air would produce a cone rather than a tight column. The wing develops lift by transferring momentum to the air. For straight and level flight this momentum eventually strikes the earth. If an airplane were to fly over a very large scale, the scale would weigh the airplane.

Let us do a back-of-the-envelope calculation to see how much air a wing might divert. Take for example a Cessna 172 that weighs about 2300 lb (1045 kg). Traveling at a speed of 140 mph (220 km/h), and assuming an effective angle of attack of 5 degrees, we get a vertical velocity for the air of about 11.5 mph (18 km/h) right at the wing. If we assume that the average vertical velocity of the air diverted is half that value we calculate from Newton's second law that the amount of air diverted is on the order of 5 ton/s. Thus, a Cessna 172 at cruise is diverting about five times its own weight in air per second to produce lift. Think how much air is diverted by a 250-ton Boeing 777.

Diverting so much air down is a strong argument against lift being just a surface effect (that is only a small amount of air around the wing accounts for the lift), as implied by the popular explanation. In fact, in order to divert 5 ton/sec the wing of the Cessna 172 must accelerate all of the air within 18 feet (7.3 m) above the wing. One should remember that the density of air at sea level is about 2 lb per cubic yard (about 1kg per cubic meter). Figure 7 illustrates the effect of the air being diverted down from a wing. A huge hole is punched through the fog by the downwash from the airplane that has just flown over it.



Figure 7. Downwash and wing vortices in the fog.

So how does a thin wing divert so much air? When the air is bent around the top of the wing, it pulls on the air above it accelerating that air downward. Otherwise there would be voids in the air above the wing. Air is pulled from above. This pulling causes the pressure to become lower above the wing. It is the acceleration of the air above the wing in the downward direction

that gives lift. (Why the wing bends the air with enough force to generate lift will be discussed in the next section.)

Normally, one looks at the air flowing over the wing in the frame of reference of the wing. In other words, to the pilot the air is moving and the wing is standing still. We have already stated that an observer on the ground would see the air coming off the wing almost vertically. But what is the air doing below the wing? Figure 8 shows an instantaneous snapshot of how air molecules are moving as a wing passes by. Remember in this figure the air is initially at rest and it is the wing moving. Arrow "1" will become arrow "2" and so on. Ahead of the leading edge, air is moving up (upwash). At the trailing edge, air is diverted down (downwash). Over the top the air is accelerated towards the trailing edge. Underneath, the air is accelerated forward slightly. Far behind the wing the air is going straight down.

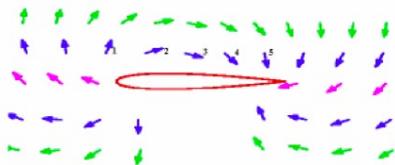


Figure 8. Direction of air movement around a wing as seen by an observer on the ground.

So, why does the air follow this pattern? First, we have to bear in mind that air is considered an incompressible fluid for low-speed flight. That means that it cannot change its volume and that there is a resistance to the formation of voids. Now the air has been accelerated over the top of the wing by of the reduction in pressure. This draws air from in front of the wing and expels it back and down behind the wing.

This air must be compensated for, so the air shifts around the wing to fill in. This is similar to the circulation of the water around a canoe paddle. This circulation around the wing is no more the driving force for the lift on the wing than is the circulation in the water drives the paddle. Though, it is true that if one is able to determine the circulation around a wing the lift of the wing can be calculated. Lift and circulation are proportional to each other.

One observation that can be made from Figure 8 is that the top surface of the wing does much more to move the air than the bottom. So the top is the more critical surface. Thus, airplanes can carry external stores, such as drop tanks, under the wings but not on top where they would interfere with lift. That is also why wing struts under the wing are common but struts on the top of the wing have been historically rare. A strut, or any obstruction, on the top of the wing would interfere with the lift.

Air Bending Over a Wing

As always, simple statements often result in more questions. One natural question is why does the air bend around the wing? This question is probably the most challenging question in understanding flight and it is one of the key concepts.

Let us start by first looking at a simple demonstration. Run a small stream of water from a faucet and bring a horizontal water glass over to it until it just touches the water, as in Figure 9. As in the figure, the water will wrap partway around the glass. From Newton's first law we know that for the flow of water to bend there must be a force on it. The force is in the direction of the bend.

From Newton's third law we know that there must be an equal and opposite force acting on the glass. The stream of water puts a force on the glass that tries to pull it into the stream, not push it away as one might first expect.

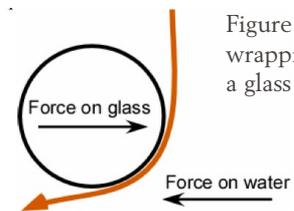


Figure 9. Water wrapping around a glass

So why does the water bend around the glass, or air over a wing? First consider low-speed flight. In low-speed flight the forces on the air and the associated pressures are so low that the air is not only considered a fluid but an incompressible fluid. This means that the volume of a mass of air remains constant and that flows of air do not separate from each other to form voids (gaps).

A second point to understand is that streamlines communicate with each other. A streamline, in steady-state flight, can be looked at as the path of a particle in the moving air. It is the path a small, light object would take in the airflow over the wing. The communication between streamlines is an expression of pressure and viscosity. Pressure is the force per area that the air exerts on the neighboring streamline. Viscosity in a gas or liquid corresponds to friction between solids.

Think of two adjacent streamlines with different speeds. Since these streamlines have different velocities forces between them trying to speed up the slower streamline and slow down the faster streamline. The speed of air at the surface of the wing is exactly zero

with respect to the surface of the wing. This is an expression of viscosity. The speed of the air increases with distance from the wing as shown in Figure 10. Now imagine the first non-zero velocity streamline that just grazes the highpoint of the top of the wing. If it were initially to go straight back and not follow the wing, there would be a volume of zero velocity air between it and the wing. Forces would strip this air away from the wing and without a streamline to replace it, the pressure would lower. This lowering of the pressure would bend the streamline until it followed the surface of the wing.



Figure 10. The variation of the speed of a fluid near an object

The next streamline above would be bent to follow the first by the same process, and so on. The streamlines increase in speed with distance from the wing for a short distance. This is on the order of 6 inch (15 cm) at the trailing edge of the wing of an Airbus A380. This region of rapidly changing air speed is the boundary layer. If the boundary layer is not turbulent, the flow is said to be laminar.

Thus, the streamlines are bent by a lowering of the pressure. This is why the air is bent by the top of the wing and why the pressure above the wing is lowered. This lowered pressure decrease with distance above the wing but is the basis of the lift on a wing. The lowered pressure propagates out at the speed of sound, causing a great deal of air to bend around the wing.

Two streamlines communicate on a molecular scale. This is an expression of the pressure and the viscosity of air. Without viscosity there would be no communication between streamlines and no boundary layer. Often, calculations of lift are made in the limit of zero viscosity. In these cases viscosity is re-introduced implicitly with the Kutta-Joukowski condition, which requires that the air come smoothly off at the trailing edge of the wing. Also, the calculations require that the air follows the surface of the wing which is another introduction of the effects of viscosity. One result of the near elimination of viscosity from the calculations is that there is no boundary layer calculated.

It should be noted that the speed of the uniform flow over the top of the wing is faster than the free-stream velocity, which is the velocity of the undisturbed air some distance from the wing. The bending of the air causes the reduction in pressure above the wing. This reduction in pressure causes an acceleration of the air. It is often taught that the acceleration of the air causes a reduction in pressure. In fact, it is the reduction of pressure that accelerates the air in agreement with Newton's first law.

Let us look at the air bending around the wing in Figure 11. To bend the air requires a force. As indicated by the colored arrows, the direction of the force on the air is perpendicular to the bend in the air. The magnitude of the force is proportional to the tightness of the bend. The tighter the air bends the greater the force on it. The forces on the wing, as shown by the black arrows in the figure, have the same magnitude as the forces on the air

but in the opposite direction. These forces, working through pressure, represent the mechanism in which the force is transferred to the wing.

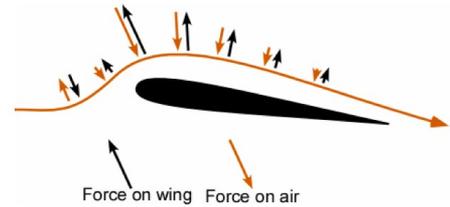


Figure 11. Forces on the air and the corresponding reaction forces on the wing

Look again at Figure 11, while paying attention to the black arrows representing the forces on the wing. There are two points to notice. The first is that most of the lift is on the forward part of the wing. In fact, half of the total lift on a wing at subsonic speeds is typically produced in the first one-fourth of the chord length. The chord is the distance from the leading edge to the trailing edge of the wing. The second thing to notice is that the arrows on the leading part of the wing are tilted forward. Thus the force of lift is pulling the wing along as well as lifting it. This would be nice if it were the entire story. Unfortunately, the horizontal forces on the trailing part of the wing compensate the horizontal forces on the leading part of the wing.

We now have the tools to understand why a wing has lift. In brief, the air bends around the wing producing downwash. Newton's first law says that the bending of the air requires a force on the air, and Newton's third law says that there is an equal and opposite force on the wing. That is a description of lift. The pressure difference across the wing is the mechanism in which lift is transferred to the wing due to the bending of the air.

Lift as a function of angle of attack

There are many types of wing: conventional, symmetric, conventional in inverted flight, the early biplane wings that looked like warped boards, and even the proverbial “barn door”. In all cases, the wing is forcing the air down, or more accurately pulling air down from above. (although the early wings did have a significant contribution from the bottom.) What each of these wings has in common is an angle of attack with respect to the oncoming air. It is the angle of attack that is the primary parameter in determining lift.

To better understand the role of the angle of attack it is useful to introduce an “effective” angle of attack, defined such that the angle of the wing to the oncoming air that gives zero lift is defined to be zero degrees. If one then changes the angle of attack both up and down one finds that the lift is proportional to the angle. Figure 12 shows the lift of a typical wing as a function of the effective angle of attack. A similar lift versus angle of attack relationship is found for all wings, independent of their design. This is true for the wing of a 747, an inverted wing, or your hand out the car window. The inverted wing can be explained by its angle of attack, despite the apparent contradiction with the popular explanation of lift. A pilot adjusts the angle of attack to adjust the lift for the speed and load. The role of the angle of attack is more important than the details of the wings shape in understanding lift. The shape comes into play in the understanding of stall characteristics and drag at high speed.

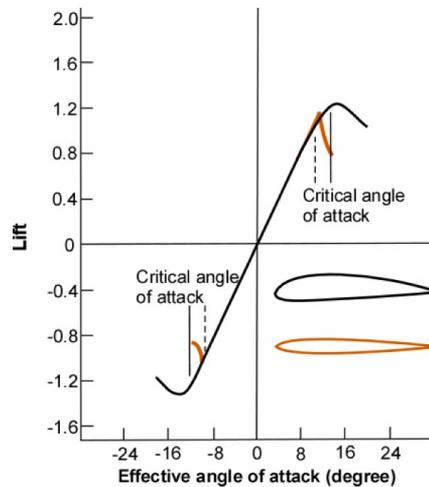


Figure 12. Lift as a function of angle of attack

One can see in the figure that the lift is directly proportional to the effective angle of attack. The lift is positive (up) when the wing is tilted up and negative (down) when it is tilted down. When corrected for area and aspect ratio, a plot of the lift as a function of the effective angle of attack is essentially the same for all wings and all wings inverted. This is true until the wing approaches a stall. The stall begins at the point where the angle of attack becomes so great that the airflow begins to separate from the trailing edge of the wing. This angle is called the critical angle of attack and is marked on the figure. This separation of the airflow from the top of the wing is a stall.

The wing as air “virtual virtual scoop”

We now would like to introduce a new mental image of a wing. One is used to thinking of a wing as a thin blade that slices through the air and develops lift somewhat by magic. For this we would like to adopt a visualization aid of looking

at the wing as a virtual scoop that intercepts a certain amount of air and diverts it to the angle of the downwash. This is not intended to imply that there is a real, physical scoop with clearly defined boundaries, and uniform flow. But this visualization aid does allow for a clear understanding of how the amount diverted air is affected by speed and density. The concept of the virtual scoop does have a real physical basis but beyond the scope of this work.

The virtual scoop diverts a certain amount of air from the horizontal to roughly the angle of attack, as depicted in Figure 13. For wings of typical airplanes it is a good approximation to say that the area of the virtual scoop is proportional to the area of the wing. The shape of the virtual scoop is approximately elliptical for all wings, as shown in the figure. Since the lift of the wing is proportional to the amount of air diverted, the lift of is also proportional to the wing’s area.

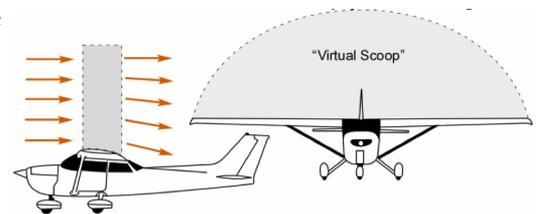


Figure 13. The “virtual scoop” as a visualization tool

As stated before, the lift of a wing is proportional to the amount of air diverted down times the vertical velocity of that air. As a plane increases speed, the virtual scoop diverts more air. Since the load on the wing does not increase, the vertical velocity of the diverted air must be decreased proportionately. Thus, the angle of attack is reduced to maintain a constant lift.

When the plane goes higher, the air becomes less dense so the virtual scoop diverts less air at a given speed. Thus, to compensate the angle of attack must be increased. The concepts of this section will be used to understand lift in a way not possible with the popular explanation.

Lift requires power

When a plane passes overhead the formally still air gains a downward velocity. Thus, the air is left in motion after the plane leaves. The air has been given energy. Power is energy, or work, per time. So, lift requires power. This power is supplied by the airplane's engine (or by gravity and thermals for a sailplane).

How much power will we need to fly? If one fires a bullet with a mass, m , and a velocity, v , the energy given to the bullet is simply $\frac{1}{2}mv^2$. Likewise, the energy given to the air by the wing is proportional to the amount of air diverted down times the vertical velocity squared of that diverted air. We have already stated that the lift of a wing is proportional to the amount of air diverted times the vertical velocity of that air. Thus, the power needed to lift the airplane is proportional to the load (or weight) times the vertical velocity of the air. If the speed of the plane is doubled, the amount of air diverted down also doubles. Thus to maintain a constant lift, the angle of attack must be reduced to give a vertical velocity that is half the original. The power required for lift has been cut in half. This shows that the power required for lift becomes less as the airplane's speed increases. In fact, we have shown that this power to create lift is proportional to $1/\text{speed}$ of the plane.

But, we all know that to go faster (in cruise) we must apply more power. So there must be more to power than the power required for lift. The power associated with lift is often called the "induced" power. Power is also needed to overcome what is called "parasite" drag, which is the drag associated with moving the wheels, struts, antenna, etc. through the air. The energy the airplane imparts to an air molecule on impact is proportional to the speed² (from $\frac{1}{2}mv^2$). The number of molecules struck per time is proportional to the speed. The faster one goes the higher the rate of impacts. Thus the parasite power required to overcome parasite drag increases as the speed³.

Figure 14 shows the "power curves" for induced power, parasite power, and total power (the sum of induced power and parasite power). Again, the induced power goes as $1/\text{speed}$ and the parasite power goes as the speed³. At low speed the power requirements of flight are dominated by the induced power. The slower one flies the less air is diverted and thus the angle of attack must be increased to increase the vertical velocity of that air. Pilots practice flying on the "backside of the power curve" so that they recognize that the angle of attack and the power required to stay in the air at very low speeds are considerable.

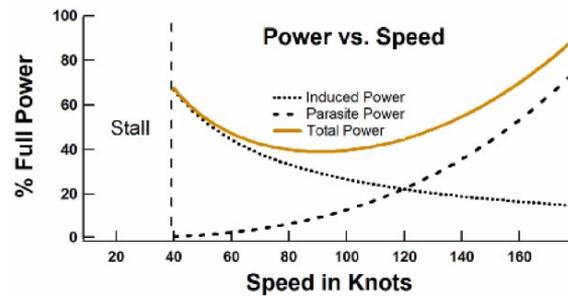


Figure 14. The power required for flight as a function of speed.

At cruise, the power requirement is dominated by parasite power. Since this goes as the speed³ an increase in engine size gives one a faster rate of climb but does little to improve the cruise speed of the plane. Doubling the size of the engine will only increase the cruise speed by about 25%.

Since we now know how the power requirements vary with speed, we can understand drag, which is a force. Drag is simply power divided by speed. Figure 14 shows the induced, parasite, and total drag as a function of speed. Here the induced drag varies as $1/\text{speed}^2$ and parasite drag varies as the speed². Taking a look at these figures one can deduce a few things about how airplanes are designed. Slower airplanes, such as gliders, are designed to minimize induced power, which dominates at lower speeds. Faster propeller-driven airplanes are more concerned with parasite power, and jets are dominated by parasite drag. (This distinction is outside of the scope of this article.)

Wing efficiency

At cruise, a non-negligible amount of the drag of a modern wing is induced drag. Parasite drag of a Boeing 747 wing is only equivalent to that of a 1/2-inch cable of the same length. One might ask what affects the efficiency of a wing. We saw that the induced power of a wing is proportional to the vertical velocity of the air. If the area of a wing were to be increased, the size of our virtual scoop would also increase, diverting more air. So, for the same lift the vertical velocity (and thus the angle of attack) would have to be reduced. Since the induced power is proportional to the vertical velocity of the air, it is also reduced. Thus, the lifting efficiency of a wing increases with increasing wing area. The larger the wing the less induced power required to produce the same lift, though this is achieved with and increase in parasite drag.

There is a misconception by some that lift does not require power. This comes from aeronautics in the study of the idealized theory of wing sections (airfoils). When dealing with an airfoil, the picture is actually that of a wing with infinite span. We have seen that the power necessary for lift decrease with increasing area of the wing. A wing of infinite span does not require power for lift since it develops lift by diverting an infinite amount of air at near-zero velocity. If lift did not require power airplanes would have the same range full as they do empty, and helicopters could hover at any altitude and load. Best of all, propellers (which are rotating wings) would not require much power to produce thrust. Unfortunately, we live in the real world where both lift and propulsion require power.

Power and wing loading

Now let us consider the relationship between wing loading and power. At a constant speed, if the wing loading is increased the vertical velocity of the downwash must be increased to compensate. This is accomplished by increasing the angle of attack of the wing. If the total weight of the airplane were doubled (say, in a 2g turn), and the speed remains constant, the vertical velocity of the air is doubled to compensate for the increased wing loading. The induced power is proportional to the load times the vertical velocity of the diverted air, which have both doubled. Thus the induced power requirement has increased by a factor of four! So induced power is proportional to the load².

One way to measure the total power is to look at the rate of fuel consumption. Figure 16 shows the fuel consumption versus gross weight for a large transport airplane traveling at a constant speed (obtained from actual data). Since the speed is constant the change in fuel consumption is due to the change in induced power. The data are fitted by a constant (parasite power) and a term that goes as the load². This second term is just what was predicted in our Newtonian discussion of the effect of load on induced power.

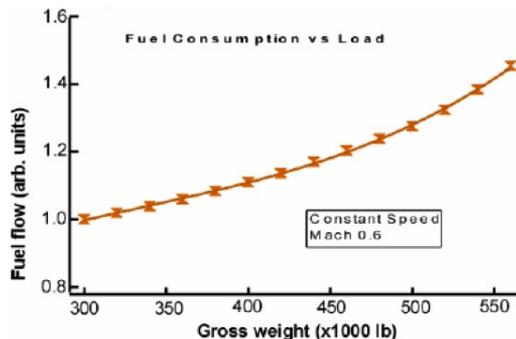


Figure 16. Fuel consumption as a function of weight for large jet at a constant speed.

The increase in the angle of attack with increased load has a downside other than just the need for more power. As shown in Figure 12 a wing will eventually stall when the air can no longer follow the upper surface. That is, when the critical angle is reached. Figure 17 shows the angle of attack as a function of airspeed for a fixed load and for a 2-g turn. The angle of attack at which the plane stalls is constant and is not a function of wing loading. The angle of attack increases as the load and the stall speed increases as the square root of the load. Thus, increasing the load in a 2-g turn increases the speed at which the wing will stall by 40%. An increase in altitude will further increase the angle of attack in a 2-g turn. This is why pilots practice “accelerated stalls” which illustrates that an airplane can stall at any speed, since for any speed there is a load that will induce a stall.

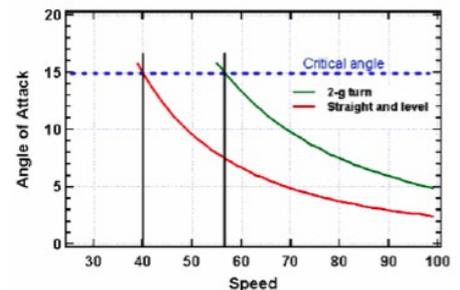


Figure 17. Angle of attack versus speed for straight and level flight and for a 2-g turn.

Wing vortices

One might ask what the downwash from a wing looks like. The downwash comes off the wing as a sheet and is related to the details on the load distribution on the wing. Figure 18 shows, through condensation, the distribution of lift on an airplane during a high-g maneuver. From the figure one can see that the distribution of load changes from the root of the wing to the tip. Thus, the amount of air in the downwash must also change along the wing. The wing near the root is “virtual scooping” up much more air than the tip. Since the wing near the root is diverting so much air the net effect is that the downwash sheet will begin to curl outward around itself, just as the air bends around the top of the wing because of the change in the velocity of the air. This is the wing vortex. The tightness of the curling of the wing vortex is proportional to the rate of change in lift along the wing. At the wing tip the lift must rapidly become zero causing the tightest curl. This is the wing tip vortex and is just a small (though often most visible) part of the wing vortex. Returning to Figure 7 one can clearly see the development of the wing vortices in the downwash as well as the wing tip vortices.



Figure 18. Condensation showing the distribution of lift along a wing.

Winglets (those small vertical extensions on the tips of some wings) are used to improve the efficiency of the wing by increasing the effective length, and thus area, of the wing. The lift of a normal wing must go to zero at the tip because the bottom and the top communicate around the end. The winglet blocks this communication so the lift can extend farther out on the wing. Since the efficiency of a wing increases with area, this gives increased efficiency. One caveat is that winglet design is tricky and winglets can actually be detrimental if not properly designed.

Ground effect

The concept of ground effect is well known to pilots. This effect is the increase in efficiency of a wing as it comes to within about a wing's length of the ground. The effect increases with the reduction in the distance to the ground. A low-wing airplane will experience a reduction in the induced drag of as much as 50 percent just before touchdown. This reduction in drag just above a surface is used by large birds, which can often be seen flying just above the surface of the water. Pilots taking off from deep-grass or soft runways also use ground effect. The pilot is able to lift the airplane off the soft surface at a speed too slow to maintain flight out of ground effect. This reduces the resistance on the wheels and allows the airplane to accelerate to a higher speed before climbing out of ground effect.

What is the cause of this reduction in drag? There are two contributions that can be credited with the reduction in drag. The ground influences the flow field around the wing which, for a given angle of attack, increases the lift. But, at the same time, there is a reduction in downwash. It can be surmised that this additional lift must come from an increase in pressure between the wing and the ground. In addition, since lift is increased for a given angle of attack, the angle of attack can be reduced for the same lift, resulting in less downwash and less induced drag.

Ground effect introduces a fundamental change from the discussion of flight at altitude. When no ground is present, the relationship between lift, drag and downwash is straight forward. But, near the ground, there is an action-reaction between the wing, the air and the ground. At altitude the ground is so distant that this effect does not exist. Near the ground this interaction helps produce lift and reduce downwash due to an increase in pressure below the wing. The details of ground effect are extremely complex. Most aerospace texts devote a paragraph or two and don't attempt to describe it in depth. The truth is that so much is changing in ground effect that it is difficult to describe by pointing to a single change in the air flow or a term in an equation. There is no simple way to describe how the airflow adjusts to satisfy the change in conditions.

Conclusions

Let us review what we have learned and get some idea of how the physical description has given us a greater ability to understand flight. First what have we learned:

- The amount of air diverted by the wing is proportional to the speed of the wing and the air density.
- The vertical velocity of the diverted air is proportional to the speed of the wing and the angle of attack.
- The lift is proportional to the amount of air diverted times the vertical velocity of the air.
- The power needed for lift is proportional to the lift times the vertical velocity of the air. Now let us look at some situations from the physical point of view and from the perspective of the popular explanation.
- The plane's speed is reduced. The physical view says that the amount of air diverted is reduced so the angle of attack is increased to compensate. The power needed for lift is also increased. The popular explanation cannot address this.
- The load of the plane is increased. The physical view says that the amount of air diverted is the same but the angle of attack must be increased to give additional lift. The power needed for lift has also increased. Again, the popular explanation cannot address this.

- A plane flies upside down. The physical view has no problem with this. The plane adjusts the angle of attack of the inverted wing to give the desired lift. The popular explanation implies that inverted flight is impossible.

As one can see, the popular explanation, which fixates on the shape of the wing, may satisfy many but it does not give one the tools to really understand flight. The physical description of lift is easy to understand and much more powerful. ■

David Anderson is a private pilot and a lifelong flight enthusiast. He has degrees from the University of Washington, Seattle, and a Ph.D. in physics from Columbia University. He has had a 30-year career in high-energy physics at Los Alamos National Laboratory, CERN in Geneva, Switzerland, and the Fermi National Accelerator Laboratory.

Scott Eberhardt is a private pilot who works in high-lift aerodynamics at Boeing Commercial Airplanes Product Development. He has degrees from MIT and a Ph.D. in aeronautics and astronautics from Stanford University. He joined Boeing in 2006 after 20 years on the faculty of the Department of Aeronautics and Astronautics at the University of Washington, Seattle.

Reprinted with permission of the original authors. First appeared in *hn.my/allstar* (allstar.fiu.edu)

This material can be found in more detail in *Understanding Flight* 1st and 2nd editions by David Anderson and Scott Eberhardt, McGraw-Hill, 2001, and 2009.

The Messy Art of UX Sketching

By PEITER BUICK

I HEAR A LOT of people talking about the importance of sketching when designing or problem-solving, yet it seems that very few people actually sketch. As a UX professional, I sketch every day. I often take over entire walls in our office and cover them with sketches, mapping out everything from context scenarios to wireframes to presentations.

Although starting a prototype on a computer is sometimes easier, it's

not the best way to visually problem-solve. When you need to ideate website layouts or mobile applications or to storyboard workflows and context scenarios, **sketching is much more efficient**. It keeps you from getting caught up in the technology, and instead focuses you on the best possible solution, freeing you to take risks that you might not otherwise take.

Many articles discuss the power of sketching and why you should do

it, but they don't go into the how or the methods involved. Sketching seems straightforward, but there are certain ways to do it effectively. In this article, we'll cover a **collection of tools and techniques** that I (and many other UX and design folks) use every day.

Sketching ≠ Drawing

Some of the most effective sketches I've seen are far from perfect drawings. Just like your thoughts and ideas, sketches are in a constant state of flux, evolving and morphing as you reach a potential solution. Don't think that you have to be able to draw in order to sketch, although having some experience with it does help.

- Sketching is an expression of thinking and problem-solving.
- It's a form of visual communication, and, as in all languages, some ways of communicating are clearer than others.
- Sketching is a skill: the more you do it, the better you'll get at it.

When evaluating your sketches, ask yourself, "How could I better communicate these thoughts?"



My desk.

Getting caught up in evaluating your drawing ability is easy, but try to separate the two. Look at your sketch as if it were a poster. What's the first thing that's read? Where is the detailed info? Remember, the eye is drawn to the area with the most detail and contrast.

Just as one's ability to enunciate words affects how well others understand them, one's ability to draw does have an **impact on how communicative a sketch** is. The good news is that drawing and sketching are skills, and the more you do them, the better you'll get.

OK, let's get started.

Work In Layers

Often when I've done a sketch, the result looks more like a collage than a sketch. Efficiency in sketching comes from working in layers.

Technique

Start with a light-gray marker (20 to 30% gray), and progressively add layers of detail with darker markers and pens.

Why?

Starting with a light-gray marker makes this easy. It allows you to make mistakes and evaluate your ideas as you work through a problem. Draw a crooked line with the light marker? No big deal. The lines will barely be noticeable by the time you're finished with the sketch.

As the pages fill up with ideas, go back in with a darker marker (60% gray) or pen, and layer in additional details for the parts you like. This is also a great way to make a particular sketch pop beside other sketches.

Sketching in layers also keeps you from **getting caught up in details right away**. It forces you to decide on the content and hierarchy of the view first. If you are sketching an interface that contains a list, but you don't yet know what will go in the list, put in a few squiggles. Later, you can go back in and sketch a few options for each list item and append them to the page.

Caution

If you start drawing with a ball-point pen and then go in later with a marker, the pen's ink will likely smear from the alcohol in the marker.

As you get more confident in your sketching, you will become more comfortable and find that you don't need to draw as many underlays. But I still find it useful because it allows you to experiment and evaluate ideas as you sketch.

Loosen Up

Technique

When sketching long lines, consider moving your arm and pen with your shoulder rather than from the elbow or wrist. Reserve drawing with your wrist for short quick lines and areas where you need more control.

Why?

This will allow you to draw longer, straighter lines. If you draw from the elbow, you'll notice that the lines all have a slight curve to them. Placing two dots on the paper, one where you want the line to start and one where you want it to end, is sometimes helpful. Then, orient the paper, make a practice stroke or two, and then draw the line.

A bonus to drawing from the shoulder is that much of the motion translates to drawing on a whiteboard; so, in time, your straight lines will be the envy of everyone in the room.

Play To Your Strengths

Technique

Rotate the page before drawing a line in order to draw multiple angles of lines more easily.

Why?

Very few people can draw lines in all directions equally well. **Rotating the page** allows you to draw a line in the range and direction that works best for you. Don't try to draw a vertical line if you find it difficult; rotate the page 90 degrees, and draw a horizontal one instead. It's super-simple but amazingly powerful.

Caution

This does not translate well to a whiteboard, so you'll still need to learn to draw vertical lines.

Sketching Interactions

Technique

Start with a base sketch, and then use sticky notes to add tooltips, pop-overs, modal windows and other interactive elements.

Why?

Using sticky notes to define tooltips and other interactive elements lets you **quickly define interactions** and concepts without having to redraw the framework of the application. They are easy to move around and can be sketched on with the same markers and pens you are already using.

- Define multiple interactions on one sketch, and then strategically remove pieces one at a time before scanning them in or copying the sketch.
- Use different colors to represent different types of interaction.

- Is one sticky note not big enough for your modal window? Add another right next to it.
- Is one sticky note too big for your tooltip, user a ruler as a guide to quickly rip the note down to size.

Copying And Pasting For The Real World

At times, you may want to manually redraw a UI element multiple times in a sketch. This is not always a bad thing, because it gives you the opportunity to quickly iterate and forces you to reconsider your ideas. That being said, an all-in-one scanner or photocopier could dramatically increase your efficiency.

Technique

Use a photocopier to quickly create templates from existing sketches or to redraw an area of a sketch.

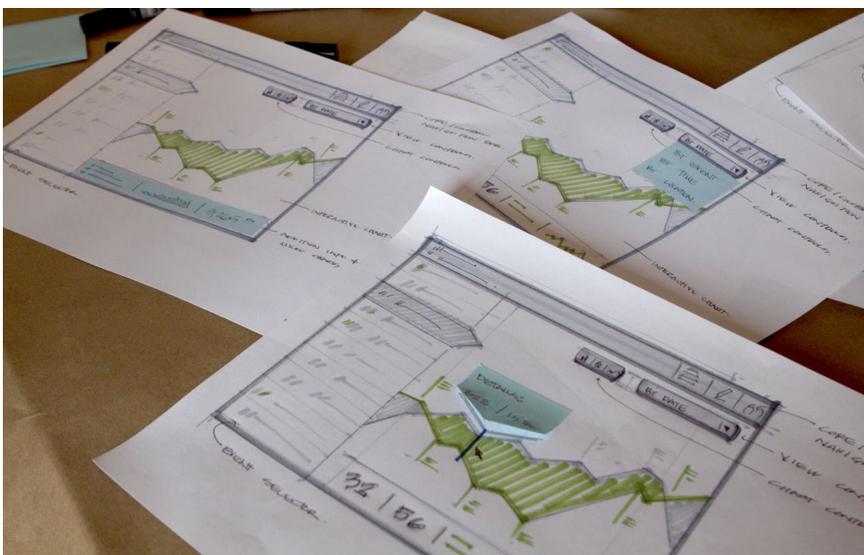
Why?

A photocopier is the old-school version of **Control + C**, **Control + V**. It makes the production of templates and underlays more efficient. It also boosts your confidence, because if you mess up (and you *will* mess up), you can easily fix it.

- Does one part of your interface need to be consistently redrawn in multiple sketches? Run a few copies, and then sketch directly on the print-outs.
- Did you mess up a part of the sketch? No problem. Cover up that portion of the sketch with a piece of paper or with correction fluid, run off a copy, and then start sketching directly on the print-out.
- Are you working on a mobile project? Or do you want to make a series of sketches all of the same size? Create a layout and copy off a few rounds of underlays. Easier yet, print off underlays of devices or browsers; a good selection can be found in the article “Free Printable Sketching, Wireframing and Note-Taking PDF Templates 8.” [hn.my/wireframe]
- Do you want to change the layout of a sidebar in your last five sketches? Sketch the new sidebar, run off a few copies, and then tape the new sidebars over the old ones. It’s that easy.
- To use a sketch as an underlay of another similar one, adjust the density or darkness setting on your photocopier to run a copy of the sketch at 20% of its original value.



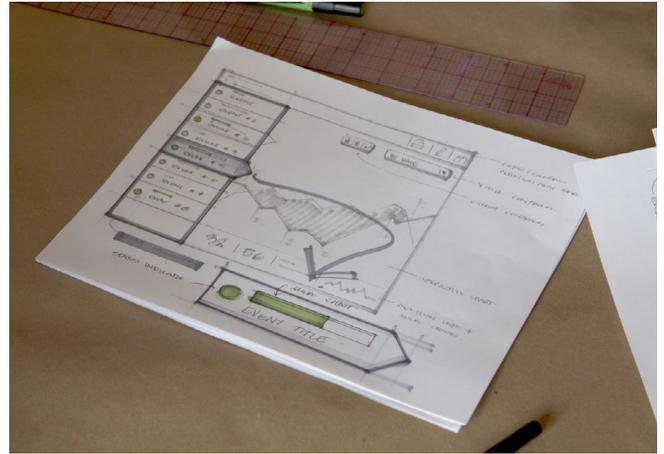
Explore a variety of interactions and ideas in a single sketch using sticky notes.



Upon photocopying various versions of a sketch, each with different sticky notes, you’ll end up with various distinct sketches.



Sketching over a photocopy of the original to reevaluate the sidebar.



The final sketch. Notice how the sidebar and its detail are darker than the photocopy. This is intentional, because it allows you to explore ideas in the context of the overall design.



Quickly creating evenly spaced lines with a quilting ruler and 30% gray marker.

The Design Is In The Details

Use a ruler; specifically, a quilting ruler. Quilting rulers are translucent and are normally printed with a grid screen, letting you see the line you're drawing relative to the rest of the sketch.

Technique

Use a ruler and a light-gray marker to draw an underlay for a detailed sketch.

Why?

This lets you quickly draw a series of lines that are offset a set distance from each other. This works great for elements such as lists items, charts, buttons and anything else that needs to be evenly spaced. It's like an analog version of "smart guides."

Technique

After using a light-gray marker to lay out a sketch, use a ruler and ballpoint pen or black marker to finalize it.

Why?

When sketching in layers, you want the final design or layout to "pop." A ruler enables you to be more precise in detailed areas and ensures that long edges are straight.

There is no shame in using a ruler. The key is knowing when to use it. Don't start sketching with a ruler; rather, bring one in when you need the detail and precision. Remember, you're sketching, not drawing.



Ripping a sticky note with a ruler.

Technique

Use a ruler to quickly rip paper or sticky notes by firmly holding the paper with one hand and ripping away the edge with the other hand.

Why?

It's quicker than grabbing scissors; you already have the ruler with you; and you can take it through airport security.

After lightly sketching an interface with a light marker, finalize it or make one area pop by using a ruler to lay down darker lines.



Sketching ideas for a mobile application in the context of where it will be used.

Tell The Whole Story

Technique

Draw the application in the context of where and how it being used, or frame it with the device it will be used on.

Why?

This forces you to think about the environment that the application will be used in, instills empathy for your users, and establishes understanding of the challenges unique to this application.

I get it. No one wants to sketch out a monitor every time they draw a wireframe. I'm not saying you have to, but a few sketches with context go a long way. Especially with mobile devices, the more context you add to a sketch, the better. Moreover, I always sketch the device for a mobile interface as an underlay, and I often try to **sketch the UI at full scale**. This forces you to deal with the constraints of the device and makes you aware of how the user may be holding the device.

Caution

Drawing the surrounding environment can be challenging and requires a higher level of sketching competency. Don't let this intimidate you. If you're not comfortable sketching the environment or you find it takes too long, use a picture as an underlay instead.



One of the many walls of sketches in our office.

Ditch The Sketchbook

Technique

Draw on 8.5 × 11" copy paper.

Why?

Sketches are for sharing. You can easily hang 8.5 × 11" sheets on a wall to share ideas with others or to see a project in its entirety. When you need to save a sketch or two, you can easily batch scan them into a computer without ripping them out of the sketchbook. Still not convinced? Copy paper is cheaper; it allows you to use sketches as underlays without photocopying; and you don't have to choose between book-bound or spiral-bound.

What Are You Waiting For?

Sketching is not reserved for designers. Developers, project managers and business analysts can get in on the fun, too. It's the best way for teams to quickly communicate, explore and share ideas across disciplines. Also, I've found that others are more receptive to give feedback and make suggestions when shown sketches than when shown printouts or screenshots.

Remember, it's about getting ideas out, reviewing those ideas and documenting them, not about creating a work of art. When evaluating your sketches, ask yourself, "How could I better communicate these thoughts?" Getting caught up in evaluating your drawing ability is easy, but try to separate the two, and know that the more you do it, the better you'll get.

It's worth repeating that sketching is the quickest way to explore and share thinking with others. It focuses you on discovering the best possible solution, without getting caught up in the technology.

Go for it! Don't get caught up in the tools. Make a mess. And have fun! ■

Peiter Buick is Senior UX Specialist at Universal Mind. He is passionate about design's ability to directly impact peoples lives. With a background in industrial design, he brings a unique perspective to the UX community.

Reprinted with permission of the original author.
First appeared in hn.my/sketch (smashingmagazine.com)
Images by Michael Kleinpaste.

Never Create Ruby Strings Longer Than 23 Characters

By PAT SHAUGHNESSY

OBVIOUSLY THIS IS an utterly preposterous statement: it's hard to think of a more ridiculous and esoteric coding requirement. I can just imagine all sorts of amusing conversations with designers and business sponsors: "No... the size of this <input> field should be 23... 24 is just too long!" Or: "We need to explain to users that their subject lines should be less than 23 letters..." Or: "Twitter got it all wrong... the 140 limit should have been 23!"

Why in the world would I even imagine saying this? As silly as this requirement might be, there is actually a grain of truth behind it: creating shorter Ruby strings is actually much faster than creating longer ones. It turns out that this line of Ruby code:

```
str = "1234567890123456789012" + "x"
```

... is executed about twice as fast by the MRI 1.9.3 Ruby interpreter than this line of Ruby code:

```
str = "12345678901234567890123" + "x"
```

Huh? What's the difference? These two lines look identical! Well, the difference is that the first line creates a new string

containing 23 characters, while the second line creates one with 24. It turns out that the MRI Ruby 1.9 interpreter is optimized to handle strings containing 23 characters or less more quickly than longer strings. This isn't true for Ruby 1.8.

Today I'm going to take a close look at the MRI Ruby 1.9 interpreter to see how it actually handles saving string values... and why this is actually true.

Not all strings are created equal

Over the holidays I decided to read through the Ruby Hacking Guide [rhg.rubyforge.org]. If you've never heard of it, it's a great explanation of how the Ruby interpreter works internally. Unfortunately, it's written in Japanese, but a few of the chapters have been translated into English. Chapter 2, one of the translated chapters, was a great place to start since it

explains all of the basic Ruby data types, including strings.

After reading through that, I decided to dive right into the MRI 1.9.3 C source code to learn more about how Ruby handles strings; since I use RVM, for me the Ruby source code is located under `~/rvm/src/ruby-1.9.3-preview1`. I started by looking at `include/ruby/ruby.h`, which defines all of the basic Ruby data types, and `string.c`, which implements Ruby String objects.

Reading the C code, I discovered that Ruby actually uses three different types of string values, which I call:

- Heap Strings
- Shared Strings
- Embedded Strings

I found this fascinating! For years I've assumed every Ruby String object was like every other String object. But it turns out this is not true! Let's take a closer look...

Heap Strings

The standard and most common way for Ruby to save string data is in the “heap.” The heap is a core concept of the C language: it’s a large pool of memory that C programmers can allocate from and use via a call to the `malloc` function. For example, this line of C code allocates a 100 byte chunk of memory from the heap and saves its memory address into a pointer:

```
char *ptr = malloc(100);
```

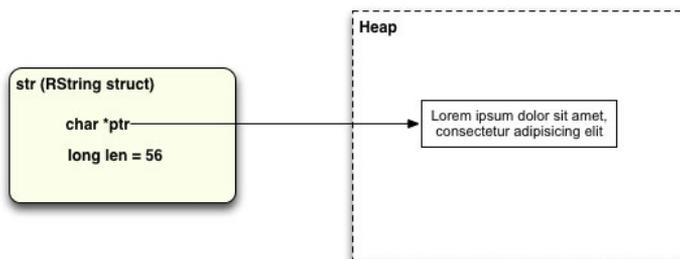
Later, when the C programmer is done with this memory, she can release it and return it to the system using `free`:

```
free(ptr);
```

Avoiding the need to manage memory in this very manual and explicit way is one of the biggest benefits of using any high level programming language, such as Ruby, Java, C#, etc. When you create a string value in Ruby code like this, for example:

```
str = "Lorem ipsum dolor  
sit amet, consectetur adip-  
iscing elit"
```

... the Ruby interpreter creates a structure called “RString” that conceptually looks like this:



You can see that the RString structure contains two values: `ptr` and `len`, but not the actual string data itself. Ruby actually saves the string character values themselves in some memory allocated from the heap, and then sets `ptr` to the location of that heap memory and `len` to the length of the string.

Here’s a simplified version of the C RString structure:

```
struct RString {  
    long len;  
    char *ptr;  
};
```

I’ve simplified this a lot; there are actually a number of other values saved in this C struct. I’ll discuss some of them next and others I’ll skip over for today. If you’re not familiar with C, you can think of `struct` (short for “structure”) as an object that contains a set of instance variables, except in C there’s no object at all – `struct` is just a chunk of memory containing a few values.

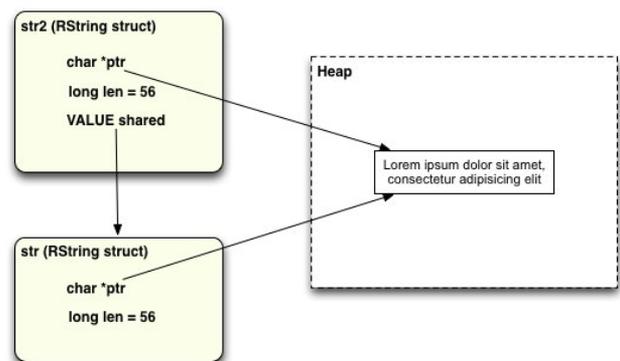
I refer to this type of Ruby string as “Heap String” since the actual string data is saved in the heap.

Shared Strings

Another type of string value that the Ruby interpreter uses is called a “Shared String” in the Ruby C source code. You create a Shared String every time you write a line of Ruby code that copies one string to another, similar to this:

```
str = "Lorem ipsum dolor sit amet, con-  
sectetur adipiscing elit"  
str2 = str
```

Here the Ruby interpreter has realized that you are assigning the same string value to two variables: `str` and `str2`. So in fact there’s no need to create two copies of the string data itself. Instead, Ruby creates two RString values that share the single copy of the string data. The way this works is that both RString structs contain the same `ptr` value to the shared data... meaning both strings contain the same value. There’s also a `shared` value saved in the second RString struct that points to the first RString struct. There are some other details, which I’m not showing here, such as some bit mask flags that Ruby uses to keep track of which RStrings are shared and which are not.



Aside from saving memory, this also speeds up execution of your Ruby programs dramatically by avoiding the need to allocate more memory from the heap using another call to `malloc`. `malloc` is actually a fairly expensive operation: it takes time to track down available memory of the proper size in the heap and also to keep track of it for freeing later.

Here's a somewhat more accurate version of the C RString structure, including the `shared` value:

```
struct RString {
    long len;
    char *ptr;
    VALUE shared;
};
```

Strings that are copied from one variable to another like this I call "Shared Strings."

Embedded Strings

The third and last way that MRI Ruby 1.9 saves string data is by embedding the characters into the RString structure itself, like this:

```
str3 = "Lorem ipsum dolor"
```

str3 (RString struct)

```
char ary[] = "Lorem ipsum dolor"
```

This RString structure contains a character array called `ary` and not the `ptr`, `len` and `shared` values we saw above. Here's another simplified definition of the same RString structure, this time containing the `ary` character array:

```
struct RString {
    char ary[RSTRING_EMBED_LEN_MAX + 1];
}
```

If you're not familiar with C code, the syntax `char ary[100]` creates an array of 100 characters (bytes). Unlike Ruby, C arrays are not objects. Instead, they are really just a collection of bytes. In C you have to specify the length of the array you want to create ahead of time.

How do Embedded Strings work?

Well, the key is the size of the `ary` array, which is set to `RSTRING_EMBED_LEN_MAX+1`. If you're running a 64-bit version of Ruby, `RSTRING_EMBED_LEN_MAX` is set to 24. That means a short string like this will fit into the RString `ary` array:

```
str = "Lorem ipsum dolor"
```

... while a longer string like this will not:

```
str = "Lorem ipsum dolor sit amet, consectetur adipiscing elit"
```

How Ruby creates new string values

Whenever you create a string value in your Ruby 1.9 code, the interpreter goes through an algorithm similar to this:

- Is this a new string value or a copy of an existing string? If it's a copy, Ruby creates a Shared String. This is the fastest option since Ruby only needs a new RString structure and not another copy of the existing string data.
- Is this a long string or a short string? If the new string value is 23 characters or less, Ruby creates an Embedded String. While not as fast as a Shared String, it's still fast because the 23 characters are simply copied right into the RString structure and there's no need to call `malloc`.
- Finally, for long string values, 24 characters or more, Ruby creates a Heap String — meaning it calls `malloc` and gets some new memory from the heap, and then copies the string value there. This is the slowest option.

The actual RString structure

For those of you familiar with the C language, here's the actual Ruby 1.9 definition of RString:

```
struct RString {
    struct RBasic basic;
    union {
        struct {
            long len;
            char *ptr;
            union {
                long capa;
                VALUE shared;
            } aux;
        } heap;

        char ary[RSTRING_EMBED_LEN_MAX + 1];
    } as;
};
```

I won't try to explain all the code details here, but here are a couple important things to learn about Ruby strings from this definition:

- The `RBasic` structure keeps track of various important bits of information about this string, such as flags indicating whether it's shared or embedded, and a pointer to the corresponding Ruby String object structure.
- The `capa` value keeps track of the "capacity" of each Heap String... it turns out Ruby will often allocate more memory than is required for each Heap String, again to avoid extra calls to `malloc` if a string size changes.
- The use of `union` allows Ruby to EITHER save the `len/ptr/capa/shared` information OR the actual string data itself.
- The value of `RSTRING_EMBED_LEN_MAX` was chosen to match the size of the `len/ptr/capa` values. That's where the 23-character limit comes from.

Here's the line of code from `ruby.h` that defines this value:

```
#define RSTRING_EMBED_LEN_MAX ((int)
(sizeof(VALUE)*3)/sizeof(char)-1)
```

On a 64-bit machine, `sizeof(VALUE)` is 8, leading to the limit of 23 characters. This will be smaller for a 32-bit machine.

Benchmarking Ruby string allocation

Let's try to measure how much faster short strings are vs. long strings in Ruby 1.9.3. Here's a simple line of code that dynamically creates a new string by appending a single character onto the end:

```
new_string = str + 'x'
```

The `new_string` value will either be a Heap String or an Embedded String, depending on how long the `str` variable's value is. The reason I need to use a string concatenation operation, the `+` `'x'` part, is to force Ruby to allocate a new string dynamically. Otherwise, if I just used `new_string = str`, I would get a Shared String.

Now I'll call this method from a loop and benchmark it:

```
require 'benchmark'
```

```
ITERATIONS = 1000000
```

```
def run(str, bench)
  bench.report("#{str.length + 1} chars") do
    ITERATIONS.times do
      new_string = str + 'x'
    end
  end
end
```

Here I'm using the benchmark library to measure how long it takes to call that method 1 million times. Now running this with a variety of different string lengths:

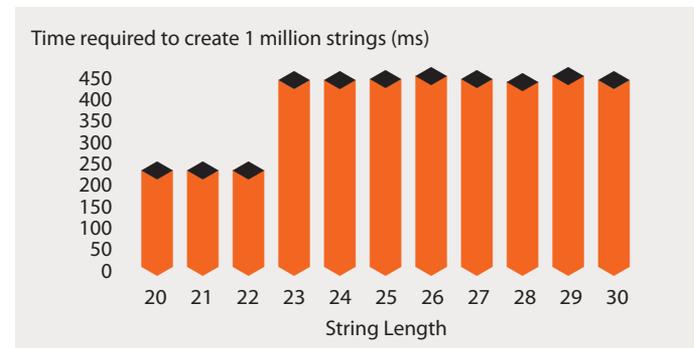
```
Benchmark.bm do |bench|
  run("12345678901234567890", bench)
  run("123456789012345678901", bench)
  run("1234567890123456789012", bench)
  run("12345678901234567890123", bench)
  run("123456789012345678901234", bench)
  run("1234567890123456789012345", bench)
  run("12345678901234567890123456", bench)
end
```

We get an interesting result:

	user	system	total	real
21 chars	0.250000	0.000000	0.250000	(0.247459)
22 chars	0.250000	0.000000	0.250000	(0.246954)
23 chars	0.250000	0.000000	0.250000	(0.248440)
24 chars	0.480000	0.000000	0.480000	(0.478391)
25 chars	0.480000	0.000000	0.480000	(0.479662)
26 chars	0.480000	0.000000	0.480000	(0.481211)
27 chars	0.490000	0.000000	0.490000	(0.490404)

Note that when the string length is 23 or less, it takes about 250ms to create 1 million new strings. But when my string length is 24 or more, it takes around 480ms, almost twice as long!

Here's a graph showing some more data; the bars show how long it takes to allocate 1 million strings of the given length:



Conclusion

Don't worry! I don't think you should refactor all your code to be sure you have strings of length 23 or less. That would obviously be ridiculous. The speed increase sounds impressive, but actually the time differences I measured were insignificant until I allocated 100,000s or millions of strings — how many Ruby applications will need to create this many string values? And even if you do need to create many string objects, the pain and confusion caused by using only short strings would overwhelm any performance benefit you might get.

For me I really think understanding something about how the Ruby interpreter works is just fun! I enjoyed taking a look through a microscope at these sorts of tiny details. I do also suspect having some understanding of how Matz and his colleagues actually implemented the language will eventually help me to use Ruby in a wiser and more knowledgeable way. ■

Pat Shaughnessy (@pat_shaughnessy) is a Ruby developer working at a global management consulting firm. Pat also writes in-depth articles at patshaughnessy.net, some of which have been featured on the Ruby Weekly newsletter, the Ruby5 podcast and the Ruby Show.



Fountain Codes

By NICK JOHNSON

FOUNTAIN CODES, OTHERWISE known as “rateless codes,” is a way to take some data — a file, for example — and transform it into an effectively unlimited number of encoded chunks, such that you can reassemble the original file given any subset of those chunks, as long as you have a little more than the size of the original file. In other words, it lets you create a “fountain” of encoded data; a receiver can reassemble the file by catching enough “droplets,” regardless of which ones they get and which ones they miss.

What makes this so remarkable is that it allows you to send a file over a lossy connection — such as, say, the internet — in a way that doesn’t rely on you knowing the rate of packet loss, and it doesn’t require the receivers to communicate anything back to you about which packets they missed. You can see how this would be useful in a number of situations, from

sending a static file over a broadcast medium, such as on-demand TV, to propagating chunks of a file amongst a large number of peers, like BitTorrent does.

Fundamentally, though, fountain codes are surprisingly simple. There are a number of variants, but for the purposes of this article, we’ll examine the simplest, called an LT, or Luby Transform Code. LT codes generate encoded blocks like this:

1. Pick a random number, d , between 1 and k , the number of blocks in the file. We’ll discuss how best to pick this number later.
2. Pick d blocks at random from the file, and combine them together. For our purposes, the xor operation will work fine.
3. Transmit the combined block, along with information about which blocks it was constructed from.

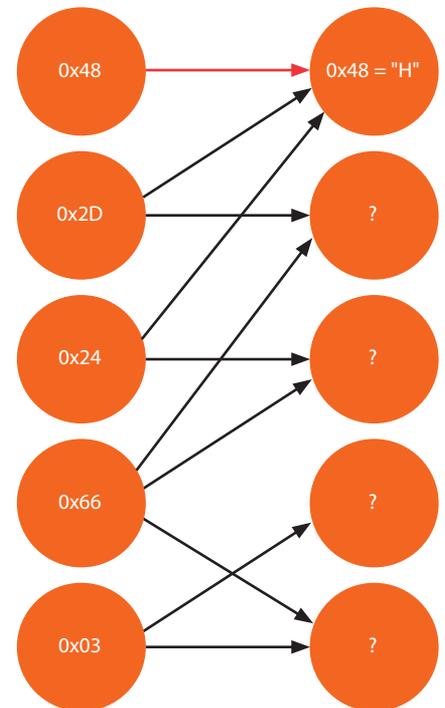
That’s pretty straightforward, right? A lot depends on how we pick the number of blocks to combine together — called the degree distribution — but we’ll cover that in more detail shortly. You can see from the description that some encoded blocks will end up being composed of just a single source block, while most will be composed of several source blocks.

Another thing that might not be immediately obvious is that while we do have to let the receiver know what blocks we combined together to produce the output block, we don't have to transmit that list explicitly. If the transmitter and receivers agree on a pseudo-random number generator, we can seed that PRNG with a randomly chosen seed and use that to pick the degree and the set of source blocks. Then, we just send the seed along with the encoded block, and our receiver can use the same procedure to reconstruct the list of source blocks we used.

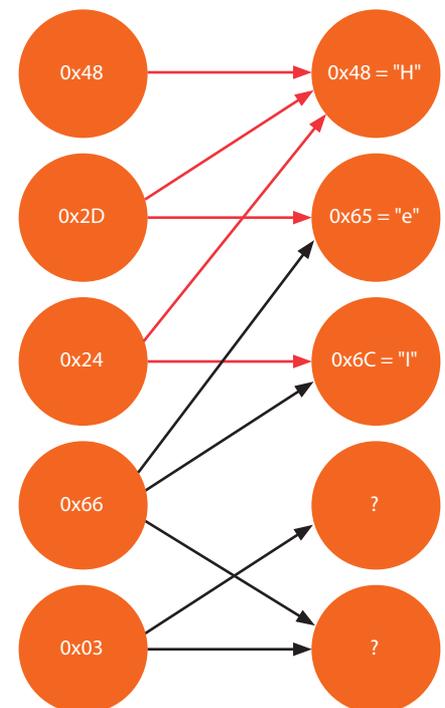
The decoding procedure is a little — but not much — more complicated:

1. Reconstruct the list of source blocks that were used to construct this encoded block.
2. For each source block from that list, xor that block with the encoded block if you have already decoded it, and remove it from the list of source blocks.
3. If there are at least two source blocks left in the list, add the encoded block to a holding area.
4. If there is only one source block remaining in the list, you have successfully decoded another source block! Add it to the decoded file, and iterate through the holding list, repeating the procedure for any encoded blocks that contain it.

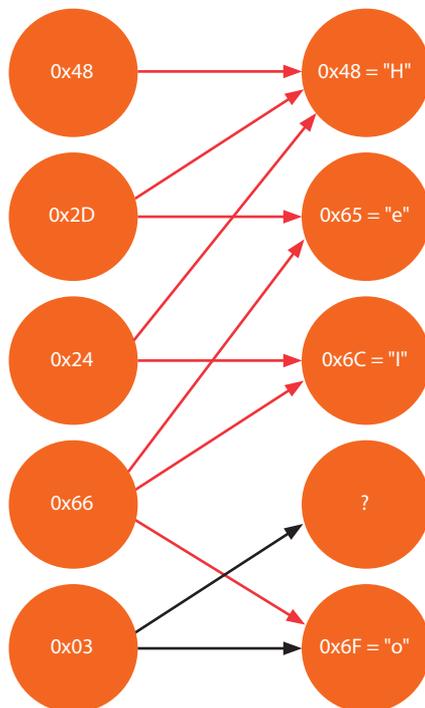
Let's work through an example of decoding to make it clearer. Suppose we receive five encoded blocks, each one byte long, along with information about which source blocks each is constructed from. We could represent our data in a graph, like this:



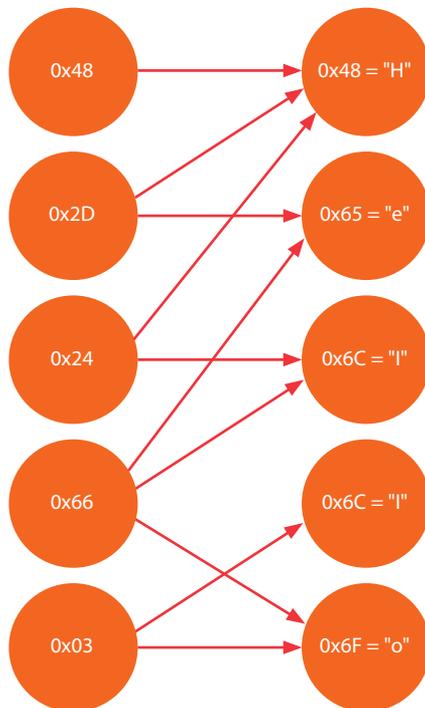
Nodes on the left represent encoded blocks we received, and nodes on the right represent source blocks. The first block we received, 0x48 turns out to consist of only one source block — the first source block — so we already know what that block was. Following the arrows pointing to the first source block, we can see that the second and third encoded blocks only depend on the first source block and one other. Since we now know the first source block, we can xor them together, giving us this:



Repeating the same procedure again, we can see we now know enough to decode the fourth encoded block, which depends on the second and third source blocks, both of which we now know. XORing them together lets us decode the fifth and final source block, giving us this:



Finally, we can now decode the last remaining source block, giving us the rest of the message:



Admittedly this is a fairly contrived example since we happened to receive just the blocks we needed to decode the message, with no extras and in a very convenient order. However, it serves to demonstrate the principle. I'm sure you can see how this applies to larger blocks and larger files quite simply.

I mentioned earlier that selecting the degree distribution, which is the number of source blocks each encoded block should consist of, is quite important. Ideally, we need to generate a few encoded blocks that have just one source block so decoding can get started, and a majority of encoded blocks that depend on a few others. It turns out such an ideal distribution exists, and is called the ideal soliton distribution.

Unfortunately, the ideal soliton distribution isn't quite so ideal in practice, as random variations make it likely that there will be source blocks that are never included, or that decoding will stall when it runs out of known blocks. A variation on the ideal soliton distribution, called the robust soliton distribution, improves on this, generating more blocks with very few source blocks and also generating a few blocks that combine all or nearly all of the source blocks to facilitate decoding the last few source blocks.

That, in a nutshell, is how fountain codes, and LT codes specifically, work. LT codes are the least efficient of the known fountain codes, but also the simplest to explain. If you're interested in learning more, I'd highly recommend reading this technical paper on fountain codes [hn.my/mackay], as well as reading about Raptor Codes [hn.my/raptor], which add only a little complexity over LT codes, but significantly improve their efficiency, both in terms of transmission overhead and computation. ■

Nick Johnson is a Developer Programs Engineer for Google App Engine, who's just seen the light and relocated to Australia. He regularly blogs about interesting computer science topics at his blog [blog.notdot.net]. When he's not saving the world there, he can be found on Twitter (@nicksdjohnson) or Stack Overflow helping folks out.

Reprinted with permission of the original author. First appeared in hn.my/fountain (notdot.net)

Unfortunate Python

By IAN WARD

PYTHON IS A wonderful language, but some parts should really have bright warning signs all over them. There are features that just can't be used safely and others are that are useful but people tend to use in the wrong ways.

Easy Stuff First

Starting with the non-controversial: Anything that has been marked deprecated should be avoided. The deprecation warning should have instructions with safe alternatives you can use.

Some of the most frequent offenders are parts of the language that make it difficult to safely call other programs:

- `os.system()`
- `os.popen()`
- `import commands`

We have the excellent `subprocess` module for these now, use it.

Ducks in a Row

Explicitly checking the type of a parameter passed to a function breaks the expected duck-typing convention of Python. Common type checking includes:

- `isinstance(x, X)`
- `type(x) == X`

With `type()` being the worse of the two.

If you must have different behavior for different types of objects passed, try treating the object as the first data type you expect, and catching the failure if that type wasn't that type, and then try the second. This allows users to create objects that are close enough to the types you expect and still use your code.

Not Really a Vegetable

```
import pickle # or cPickle
```

Objects serialized with `pickle` are tied to their implementations in the code at that time. Restoring an object after an underlying class has changed will lead to undefined behavior. Unserializing pickled data from an untrusted source can lead to remote exploits. The pickled data itself is opaque binary that can't be easily edited or reviewed.

This leaves only one place where `pickle` makes sense — short-lived data being passed between processes, just like what the `multiprocessing` module does.

Anywhere else, use a different format. Use a database or use `JSON` with a well-defined structure. Both are restricted to simple data types and are easily verified or updated outside of your Python script.

Toys are for Children

Many people are drawn to these modules because they are part of Python's standard library. Some people even try to do serious work with them.

- `asyncore / asynchat`
- `SimpleHTTPServer`

The former resembles a reasonable asynchronous library, until you find out there are no timers. At all. Use Twisted instead; it's the best we've got.

The latter makes for a neat demo by giving you a web server in your pocket with the one command `python -m SimpleHTTPServer`. But this code was never intended for production use and certainly not designed to be run as a public web server. There are plenty of real, hardened web servers out there that will run your Python code as a WSGI script. Choose one of them instead.

Foreign Concepts

```
import array
```

All the flexibility and ease of use of C arrays, now in Python!

If you really, really need this you will know. Interfacing with C code in an extension module is one valid reason.

If you're looking for speed, try just using regular Python lists and PyPy. Another good choice is NumPy for its much more capable array types.

Can't be Trusted

```
def __del__(self):
```

The mere existence of this method makes objects that are part of a reference cycle uncollectable by

Python's garbage collector and could lead to memory leaks.

Use a `weakref.ref` object with a callback to run code when an object is being removed instead.

Split Personality

```
reload(x)
```

It looks like the code you just changed is there, except the old versions of everything are still there too. Objects created before the reload will still use the code as it was when they were created, leading to situations with interesting effects that are almost impossible to reproduce.

Just re-run your program. If you're debugging at the interactive prompt, consider debugging with a small script and `python -i` instead.

Almost Reasonable

```
import copy
```

The `copy` module is harmless enough when used on objects that you create and you fully understand. The problem is once you get in the habit of using it, you might be tempted to use it on objects passed to you by code you don't control.

Copying arbitrary objects is troublesome because you will often copy too little or too much. If this object has a reference to an external resource, it's unclear what copying that even means. It can also easily lead to subtle bugs introduced into your code by a change outside your code.

If you need a copy of a list or a dict, use `list()` or `dict()` because you can be sure what you will get after they are called. `copy()`, however, might return anything, and that should scare you.

Admit You Always Hated It

```
if __name__ == '__main__':
```

This little wart has long been a staple of many Python introductions. It lets you treat a Python script as a module or a module as a Python script. Clever, sure, but it's better to keep your scripts and modules separate in the first place.

If you treat a module like a script, and then something imports the module, you're in trouble: now you have two copies of everything in that module.

I have used this trick to make running tests easier, but `setuptools` already provides a better hook for running tests. For scripts, `setuptools` has an answer too: just give it a name and a function to call, and you're done.

My last criticism is that a single line of Python should never be 10 alphanumeric characters and 13 punctuation characters. All those underscores are there as a warning that some special non-obvious language-related thing is going on, and it's not even necessary.

Don't Emulate `stdlib`

If it's in standard library, it must be well written, right?

May I present the implementation of `namedtuple`, which is a really handy little class that, if used properly, can significantly improve your code's readability:

```
def namedtuple(typename,
              field_names, verbose=False,
              rename=False):
    # Parse and validate the field
    # names. Validation serves
    # two purposes, generating
    # informative error messages
    # and preventing template
    # injection attacks.
```

Wait, what? “preventing template injection attacks”?

This is followed by 27 lines of code that validates `field_names`. And then:

```
template = '''class %(typename)s(tuple):
    %(typename)s(%(argtxt)s) \n
    __slots__ = () \n
    _fields = %(field_names)r \n
    def __new__(_cls, %(argtxt)s):
        'Create new instance of %(typename)s(%(argtxt)s)'
        return tuple.__new__(_cls, %(argtxt)s) \n
    @classmethod
    def _make(cls, iterable, new=tuple.__new__, len=len):
        'Make a new %(typename)s object from a sequence or
iterable'
        result = new(cls, iterable)
        if len(result) != %(numfields)d:
            raise TypeError('Expected %(numfields)d arguments,
got %d' %% len(result))
        return result \n
    def __repr__(self):
        'Return a nicely formatted representation string'
        return '%(typename)s(%(reprtxt)s)' %% self \n
    def _asdict(self):
        'Return a new OrderedDict which maps field names to
their values'
        return OrderedDict(zip(self._fields, self)) \n
    __dict__ = property(_asdict) \n
    def _replace(_self, **kwds):
        'Return a new %(typename)s object replacing specified
fields with new values'
        result = _self._make(map(kwds.pop, %(field_names)r,
_self))
        if kwds:
            raise ValueError('Got unexpected field names:
%%r' %% kwds.keys())
        return result \n
    def __getnewargs__(self):
        'Return self as a plain tuple. Used by copy and
pickle.'
        return tuple(self) \n\n''' % locals()
```

Yes, that’s a class definition in a big Python string, filled with variables from `locals()`. The result is then `execed` in the right namespace, and some further magic is applied to “fix” `copy()` and `pickle()`.

I believe this code was meant as some sort of warning to people that would contribute code to Python — something like “We make it look like we know what we’re doing, but we’re really just nuts” (love ya Raymond).

Trying Too Hard

```
hasattr(obj, 'foo')
```

`hasattr` has always been defined to swallow all exceptions, even ones you might be interested in (such as a `KeyboardInterrupt`), and turn them into a `False` return value. This interface just can’t be fixed, so use `getattr` with a sentinel value instead.

Off by One

```
'hello'.find('H')
```

`str.find` and `str.rfind` return `-1` on failure. This can lead to some really hard-to-find bugs when combined with containers like strings that treat `-1` as the last element. Use `str.index` and `str.rindex` instead. ■

Ian Ward is an independent software developer in Ottawa, Canada. He works primarily with Linux, Python, C and PostgreSQL. He is also the author and maintainer of the Python console user interface library `Urwid`.

Reprinted with permission of the original author.
First appeared in *hn.my/unfortunate* (excess.org)

Tips for Remote Unix Work

By BRANDON MINTERN

IF YOU ARE anything like me, you have programs running on all kinds of different servers. You probably have a GitHub account, a free Heroku instance, a work desktop, a couple website instances, and maybe even a home server. The best part is that using common Unix tools, you can connect to all of them from one place.

In this article, I will review some of the more interesting aspects of my workflow, covering the usage of SSH, screen, and VNC, including a guide for getting started with VNC. I'll provide some quick start information and quickly progress to advanced topics (like SSH pipes and auto-session-creation) that even experienced Unix users may not be aware of.

SSH to rule them all

By now you've almost certainly used SSH. It's the easiest way to login to a remote machine and get instant command line access. It's as easy as `ssh user@example.com`. You type in your password, and you're in! But you might not know that it can be even easier (and more secure) than that.

Logging in via SSH without a password

We have only recently seen websites start to offer solutions for logging in without a password. SSH has provided a secure mechanism for this (based on public-key cryptography) since its inception. It's pretty easy to setup once you know how it works.

1. Generate a public-private key pair

If you haven't already, run `ssh-keygen` on your laptop, or whatever computer you will be doing your work from. You can just continue pressing Enter to accept the defaults, and you can leave the password blank (if you secure your laptop with encryption, a locking screensaver, and a strong password, your SSH key doesn't require a password). This will generate a public key at `~/.ssh/id_rsa.pub` and a private key at `~/.ssh/id_rsa`. **The private key should never leave your computer.**

2. Copy the public key to each computer you connect to

For each computer that you connect to, run the following command:

```
ssh-copy-id user@example.com
```

(Note that you can specify `-p PORT` or any other SSH arguments before the `user@example.com` portion of the above command.)

This should be the last time you ever have to type your login password when connecting to the remote server. From now on, when you SSH to the remote server, its `sshd` service will encrypt some data using the public key that you appended to `authorized_keys`, and your local machine will be able to decode that challenge with your private key.

3. There is no step 3

It's that easy! Don't you wish you had set this up a long time ago?

SSH and pipes

If you take a look at the `ssh-copy-id` script, you'll see a line that roughly translates to:

```
cat ~/.ssh/id_rsa.pub | ssh
user@example.com "umask 077;
test -d ~/.ssh || mkdir
~/.ssh ; cat >> ~/.ssh/
authorized_keys"
```

When you ran `ssh-copy-id` above, here's what that line did:

1. The contents of `~/.ssh/id_rsa.pub` were piped into the SSH command.
2. SSH encrypted that data and sent it across the network to your remote machine.
3. Everything in double quotes after the host is a single argument to `ssh`; this specified that instead of giving you an interactive login, you instead wanted to run a command.
4. The first portion of that command (`umask 077; test -d ~/.ssh || mkdir ~/.ssh ;`) created a `.ssh` directory on the remote machine if it did not already exist, ensuring that it had the proper permissions.
5. The second portion (`cat >> ~/.ssh/authorized_keys`) received the standard input via the SSH tunnel and appended it to the `authorized_keys` file on the remote machine.

This avoids the need to use SCP and login multiple times. SSH can do it all! Here are some more examples to show you some of the neat things you can do with SSH pipe functionality.

Send the files at `~/src/` to `example.com:~/src/` without `rsync` or `scp`

```
cd && tar czv src | ssh exam-
ple.com 'tar xz'
```

Copy the remote website at `example.com:public_html/example.com` to `~/backup/example.com`

```
mkdir -p ~/backup/
```

```
cd !$
```

```
ssh example.com 'cd public_html
&& tar cz example.com' | tar
xzv
```

See if `httpd` is running on `example.com`

```
ssh example.com 'ps ax | grep
[h]ttpd'
```

Other SSH tunnels

If piped data were the only thing that could be securely tunneled over SSH connections, that would still be useful. But SSH can also make remote ports seem local. Let's say that you're logged into `example.com`, and you're editing a remote website that you'd like to test on port 8000. But you don't want just anyone to be able to connect to `example.com:8000`, and besides, your firewall won't allow it. What if you could get a connection to `example.com, localhost:8000`, but from your local computer and browser? Well, you can!

Create an SSH tunnel

```
ssh -NT -L 9000:localhost:8000
example.com
```

Using the `-L` flag, you can tell SSH to listen on a local port (9000), and to reroute all data sent and received on that port to `example.com:8000`. To any process listening on `example.com:8000`, it will look like it's talking to a local process (and it is: an SSH process). So open a terminal and run the above command, and then fire up your browser locally and browse to `localhost:9000`. You will be whisked away to `example.com:8000` as if you were working on it locally!

Let me clarify the argument to `-L` a bit more. The bit before the colon is the port on your local machine that you will connect to in order to be tunneled to the remote port. The part after the second colon is the port on the remote machine. The "localhost" bit is the remote machine you will be connected to, from the perspective of `example.com`. When you realize the ramifications of this, it becomes even more exciting! Perhaps you have a work computer to which you have SSH access, and you have a company intranet site at `192.168.10.10`. Obviously, you can't reach this from the outside. Using an SSH tunnel, however, you can!

```
ssh -NT -L
8080:192.168.10.10:80 work-
account@work-computer.com
```

Now browse to `localhost:8080` from your local machine, and smile as you can access your company intranet from home with your laptop's browser, just as if you were on your work computer.

But my connection sucks, or, GNU screen

Have you ever started a long-running command, checked in on it periodically for a couple hours, and then watched horrified as your connection dropped and all the work was lost? Don't let it happen again. Install GNU screen on your remote machine, and when you reconnect you can resume your work right where you left off (it may have even completed while you were away).

Now, instead of launching right into your work when you connect to your remote machine, first start up a screen session by running `screen`. From now on, all the work you are doing is going on inside screen. If your connection drops, you will be detached from the screen session, but it will continue running on the remote machine. You can reattach to it when you log back in by running `screen -r`. If you want to manually detach from the session but leave it running, type `Ctrl-a, d` from within the screen session.

Using screen

Screen is a complex program, and going into everything it can do would be a series of articles. Instead, check out this great screen quick reference guide [hn.my/screen]. Some of screen's more notable features are its ability to allow multiple terminal buffers in a single screen session and its scrollbar buffer.

What happened to Control-a?

Screen intercepts Control-a to enable some pretty cool functionality. Unfortunately, you may be used to using Control-a for readline navigation. You can now do this by pressing `Ctrl-a, a`. Alternatively, you can remap it by invoking screen with the `-e` option. For example, running `screen -e ^jj` would cause Control-j to be intercepted by screen instead of Control-a. If you do this, just replace references to "C-a" in the aforementioned reference guide with whatever escape key you defined.

Shift-PageUp is broken

Like `vim` and `less`, screen uses the terminal window differently from most programs, controlling the entire window instead of just dumping text to standard output and standard error. Unfortunately, this breaks Shift-PageUp and Shift-PageDown in `gnome-terminal`. Fortunately, we can fix this by creating a `~/.screenrc` file with the following line in it:

```
termcapinfo xterm ti:te@
```

And while you're mucking around in `.screenrc`, you might as well add an escape `^jj` line to it, so that you can stop typing in `-e ^jj` every time you invoke screen.

Starting screen automatically

It's pretty easy to forget to run screen after logging in. Personally, any time I am using SSH to login and work interactively, I want to be in a screen session. We can combine SSH's ability to run a remote command upon login with screen's ability to reconnect to detached sessions. Simply create an alias in your `~/.bashrc` file:

```
alias sshwork='ssh -t work-username@my-work-computer.com "screen -dR"'
```

This will automatically fire up a screen session if there is not one running, and if there is one running, it will connect to it. Detaching from the screen session will also logout of the remote server.

Remote graphical work

Even in spite of SSH's port forwarding capabilities, we still sometimes need to use graphical applications. If you have a fast connection or a simple GUI, passing the `-Y` flag to SSH could be enough to allow you run the application on your local desktop. Unfortunately, this often is a very poor user experience, and it does not work well with screen (a GUI application started in a screen session dies when you detach from the screen session).

The time-tested Unix solution to this problem is VNC. This is effectively a combination of screen and a graphical environment. Unfortunately, it has several drawbacks.

- It can be tricky to setup reasonably.
- It is inherently insecure, with unencrypted data and a weak password feature.
- Its performance on a sub-optimal connection is less-than-stellar.
- It doesn't transfer sounds over the network.

I'm going to help you solve all of these problems, except the sound one. Who needs sounds, anyway?

VNC installation and setup

On the remote machine, you'll need to install a VNC server and a decent lightweight window manager. I chose fluxbox and x11vnc:

```
sudo apt-get install x11vnc fluxbox
```

The programs that are started when you first start a VNC session are controlled by the `~/.vnc/xstartup` file. I prefer something a bit better than the defaults, so mine looks like this:

```
#!/bin/sh
[ -x /etc/vnc/xstartup ] &&
exec /etc/vnc/xstartup
[ -r $HOME/.Xresources ] &&
xrdb $HOME/.Xresources
netbeans &
gnome-terminal &
fluxbox &
```

Modify this to suit your own needs. I only invoke netbeans because it's the only reason I ever use a remote GUI at all. NB: Although it may seem counterintuitive, it's typically best to put the window manager command last.

You can start a VNC server with the following command:

```
vncserver -geometry WIDTHxHEIGHT
```

where `WIDTHxHEIGHT` is your desired resolution. For me, it's `1440x900`. The first time you run this, it will ask you to create a password. We are going to ensure security through other means, so you can set it to whatever you want. Running the above command will give a message like "New 'remote-machine:1 (username)' desktop is remote-machine:1". The `:1` is the display number. By adding `5900` to this, we can determine which port the VNC server is listening on. At this point, we can connect to `remote-machine:5901` with a `vncviewer` and log in to the session we've created. We don't want the entire Internet to be able to connect to our poorly-secured session, so let's terminate that VNC server session:

```
vncserver -kill :1
```

Securing the VNC server

Remember how we tunneled ports with SSH? We can do the same thing with VNC data. First, we'll invoke our VNC server slightly differently:

```
vncserver -localhost -geometry WIDTHxHEIGHT -SecurityTypes None
```

This causes the VNC server to only accept connections that originate on the local machine. It also indicates that we will not need a password to connect to our session; simply being logged in locally as the user who created the session is enough. You should now have a VNC server running on a remote machine listening on `localhost:5901`.

On your local machine, install a VNC viewer. I personally use `gvncviewer`, though I don't particularly recommend it. Now, to connect to that remote port, you'll need to start an SSH tunnel on your local machine:

```
ssh -NT -L 5901:localhost:5901 remote-machine.com
```

We can now run the VNC viewer on our local machine to connect via the tunnel to our VNC session:

```
gvncviewer :1
```

Speeding up VNC?

When starting an SSH tunnel, we can compress the data it sends by including the `-C` flag. Depending on your connection speed, it may be worth including the flag in your tunnel command. Experiment with this option and see what works best for you.

If you are really having problems, you might also want to check out the `-deferUpdate` option, which can delay how often display changes are sent to the client. For more information, `man Xvnc`.

Automatically starting and connecting to your VNC session

Putting everything together, we can create a script that does all of this for us. Simply set the `GEOMETRY` and `SSH_ARGS` variables appropriately (or modify it slightly to accept them as command line arguments).

```
#!/bin/bash
set -e

GEOMETRY=1440x900
SSH_ARGS='-p 22 username@remote-server.com'

# Get VNC display number. If there is not a VNC
# process running, start one
vnc_display="$(ssh $SSH_ARGS 'ps_text="$(ps x |
grep X[v]nc | awk '""''{ print $6 }'""'' | sed
s/://)"; if [ "$ps_text" = "" ]; then vncserver
-localhost -geometry '$GEOMETRY' -Security-
Types none 2>&1 | grep New | sed '""''s/^.*:\
([^\:]*\)$/\1/'""''; else echo "$ps_text"; fi)"
port=`expr 5900 + $vnc_display`
ssh -NTC -L $port:localhost:$port $SSH_ARGS &
SSH_CMD=`echo $!`
sleep 3
gvncviewer :$vnc_display
kill $SSH_CMD
```

The `vnc_display` line is pretty gross, so I'll give some explanation. It uses SSH to connect to the remote server and look for a running process named `xvnc`: this is the running VNC server. If there's one running we extract the display number. Otherwise, we start one up with the specified geometry and grab the display number

from there. This all happens within a single command executed by ssh, and the resulting output is piped across the network back into our `vnc_display` variable.

Either way we get the value, we now know which port to connect to in order to reach our VNC server. We start our SSH tunnel and get the resulting PID. Finally, we invoke the `vncviewer` on that tunneled local port. When the VNC viewer exits, we automatically kill our SSH tunnel as well.

Concluding remarks

One of the best parts of Unix is that it was built to be run remotely from Day 1. Just about anything you can do on your local computer can also be done on a remote one. By leveraging tools like SSH, screen, and VNC, we can make remote work as easy and convenient as local work. I hope this gave you some ideas for how you can create a productive workflow with these very common Unix tools. ■

Brandon Mintern is Lead Software Engineer at EasyESI, a seed-funded startup in Berkeley. His pursuits include reverse engineering, data processing, and language design. He presented at the first annual PyOhio. He currently enjoys exploring all the Bay Area has to offer.

Reprinted with permission of the original author.
First appeared in hn.my/remotunix (brandonmintern.com)

Being a Great Coder By TIMOTHY DALY

DO YOURSELF A favor and lose the “great coder” meme. Or get a job at Google and remain blissfully unaware.

One of the best books I've ever read about programming is called “Practicing: A Musician's Return to Music,” where the author talks about his development as a musician. He would receive compliments on how great he was at playing the guitar. At one point he replies, “How would you know?” The better he got, the worse he knew he was.

Your opinion of how great you are at programming will follow a bell curve. You'll start off coming out of college thinking you're ok, memorize a few algorithms and order theory (“the Google disease”) and think you're “great” (“Google only hires great coders”). But as you learn more you'll discover that you have SO much more to learn, and as you work on larger projects you'll discover the musician's insight. People would rate you “great,” but you'll be able to say, “How would you know?” At which point, the better you get, the worse you'll know you are.

Anybody who rates themselves as “great” is probably on the uphill side of the learning curve.

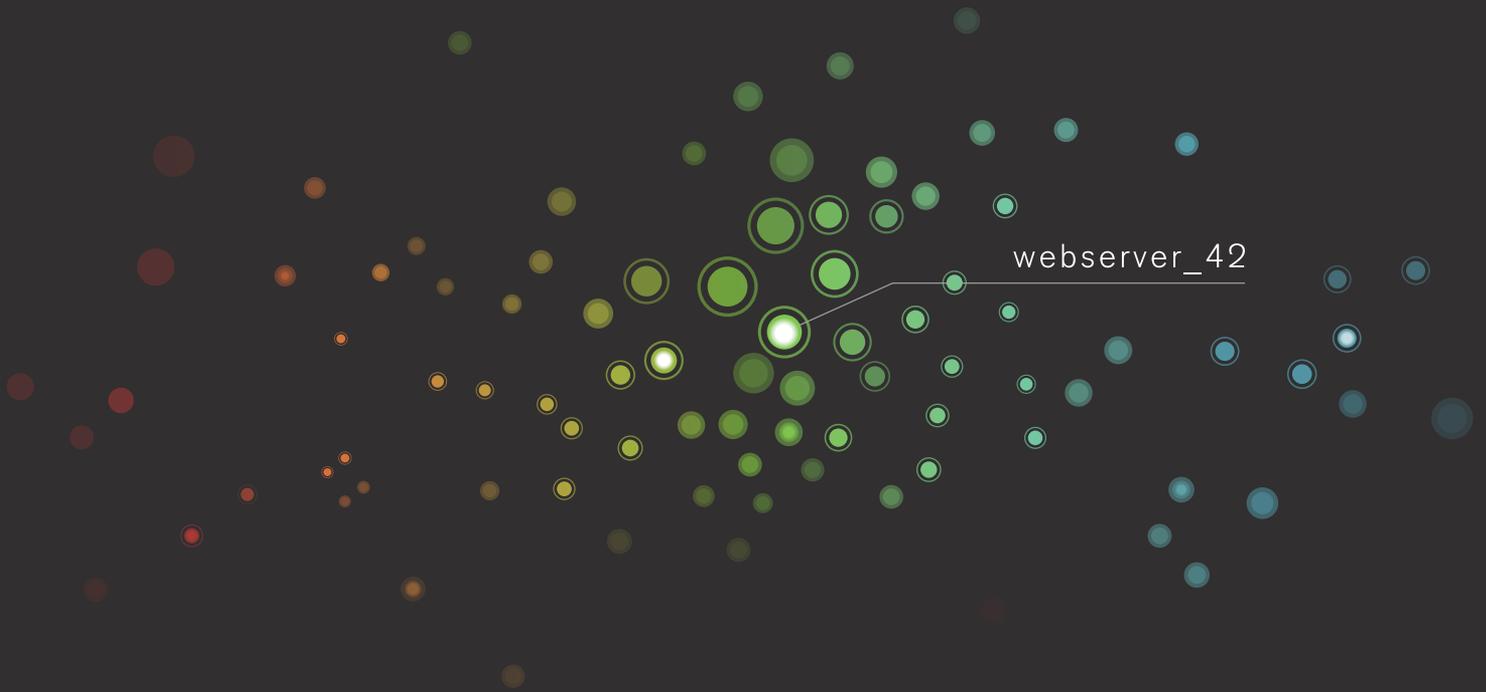
If you're trying to learn Clojure, moving into areas that are beyond your comfort zone, and trying to learn literate programming to improve your game, all points to the fact that you will likely reach a point where you feel that being labeled “great” is a sign that the speaker is clueless. Give it 10000 hours. ■

Timothy Daly is Axiom's lead developer. He is currently running his own consulting business, Literate Software, while building a base of literate tools.

These are your servers



These are your servers on Cloudkick



webserver_42

Any questions?

cloudkick.com

415.779.5425

support for 8 clouds + dedicated hardware

cloudkick

the best way to manage the cloud

Why 13th

Chords



By JAMES TAUBER

AS THE BACKGROUND to my music theory is more classical in nature, it used to puzzle me when I saw jazz chords like C9, B \flat 11 or F13. I mean, I knew what a 9th, 11th and 13th note were, but I wondered why you'd call a note a 9th rather than a 2nd, or a 13th rather than a 6th and so on.

After all, when you talk about chord, you're normally talking about notes independent of octave. If you describe something as a C7 chord, you're not saying anything about whether the E and B \flat are in the same octave or not.

I can't remember when, but the breakthrough came when I realized that a 9th chord isn't just a major triad with the 2nd added, but one with the 2nd and 7th added. An 11th chord is one with the 4th and 7th added.

(Just as an aside: the fact $2+7=9$ and $4+7=11$ here is an unrelated coincidence. An 11th is $4\text{th}+\text{octave}$, but due to the 1-based indexing used, you add 7, not 8.)

Now yes, I've seen the theory books where they show a C9 as C+E+G+B \flat +D, a C11 as C+E+G+B \flat +D+F and a C13 as C+E+G+B \flat +D+F+A, but that really didn't help emphasize that it's the existence of the 7th that makes the chord sound like (and be described as) a C9, C11 or C13 respectively, instead of, say a Cadd2, Cadd4 or C6.

The 3rd and 7th are really the defining notes of a chord in jazz, particularly comping on piano where you expect the bass to provide the root. So the final light went off when I saw the closing jazz riff of Ben Folds Five's *Underground* notated. There were a bunch of triads that were marked as 13th chords. So, for example, the voicing E \flat +A+D was marked as F13.

Note that that voicing has just the 3rd, 7th and 13th. The 13th is also a 6th, but by calling the chord F13, it's making it clear the 7th is there as well, which gives the chord a very different direction it wants to go. The 7th makes the whole chord want to resolve to a B \flat , which gives the 13th/6th (the D) more of a suspended feel it doesn't have in an F6 chord.

I find not only the 13th chord a great substitute for a 7th now, especially when it's the dominant resolving to the tonic, but I also love the 7th+3rd+13th/6th way of voicing it too.

I know this is jazz 101, but it was a breakthrough moment for me. ■

James Tauber is the founder and CEO of web startup Eldarion, Inc. When not working with software startups, websites and open source software, he is an aspiring composer, music theorist, mathematician and linguist. James lives just outside of Boston with his wife but is originally from Perth, Western Australia.



Google tracks you. We don't.

IN PHILADELPHIA, I spent a lot of time waiting for elevators. I inevitably paid a lot of attention to the control algorithms used by different elevators in different buildings.

Elevator Algorithms

By LISA ZHANG

All elevator algorithms solve the same type of optimization problem: if a building has n floors and m elevators, how could we most efficiently move people up/down the floors? I'm sure you already know of the simple algorithm that every elevator implements, but one can definitely improve on this. Here's one improvement someone tried to make:

Example #1

This building has 1 elevator and 8 floors. The elevator was made to move back to floor 4 when it is idle.

This is an intuitive solution. Since there are n floors from where people could call the elevator, why not minimize the wait time by making the elevator go back to floor $n/2$ when it is idle? The problem with this argument is that it assumes an elevator is equally likely to be called from any of the n floors, which is not true. In most cases, people who use the elevator would use it to either go down to

ground floor from the floor they're at or up from ground floor to the floor they should be in. This means that approximately half the time, elevator requests would occur at the ground floor. A better design is the following:

Example #2

There are no more than 10 floors (I believe it was less) and about 6 elevators. When an elevator is idle, it moves to the ground floor and opens its door.

This speeds things up a lot. Not only could you avoid

waiting for the elevator to get to the ground floor, you don't even have to press the button and wait for the door to open! I thought this was a great idea! An acquaintance pointed out, though, that unsuspecting people might mistakenly think the elevator is broken. Well then...

The algorithm used in Example #2 focuses a lot more on people going up rather than people going down. I think this makes sense. Going up stairs takes a lot more effort than going down stairs, so people are more likely to use the elevator to go up. However, in a building with more floors, more people would want to use the elevator to go down, so having all the elevators on ground floor is not going to help. Here's a solution that seems to work well:

Example #3

This building has 2 elevators and ~12 floors. It is programmed to ensure that at least 1 elevator is on the ground floor at any given time. The other elevator is often seen on

floor 6, but I'm not sure if there's a pattern here.

This makes a lot of sense. The first elevator takes care of the case where people want to go up from floor 1. The second elevator takes care of the case where people would want to go down, and since the elevator is at floor 6, the wait time is reduced.

For small n and m , I really can't think of a better solution than the one used in Example #3. For larger n and m , though, it becomes more complicated:

Example #4

This building has about 38 floors and at least 12 elevators. The elevators are divided into 2 groups: the first group goes to floors up to 22. The second elevator skips all the floors until floor 22, so it stops at floors 22-38 (and the ground floor).

It would be quite disastrous if elevators aren't organized this way. Imagine working on the top floor and having to wait for the elevator to stop at every floor in between! This elevator is designed to go super fast from floor 1 to floor 22, making things even more efficient.

All of these examples are real. What I don't understand is why so many buildings do not have these optimizations built into their elevators. Implementing these changes cost almost nothing, and they can save a lot of peoples' time in the long run. ■

Lisa is a pure math and applied math student at the University of Waterloo. She is passionate about data science, data mining, data visualization and entrepreneurship.

Reprinted with permission of the original author. First appeared in hn.my/elevator (lisazhang.ca)

HARVEST



TIMESHEETS



INVOICES



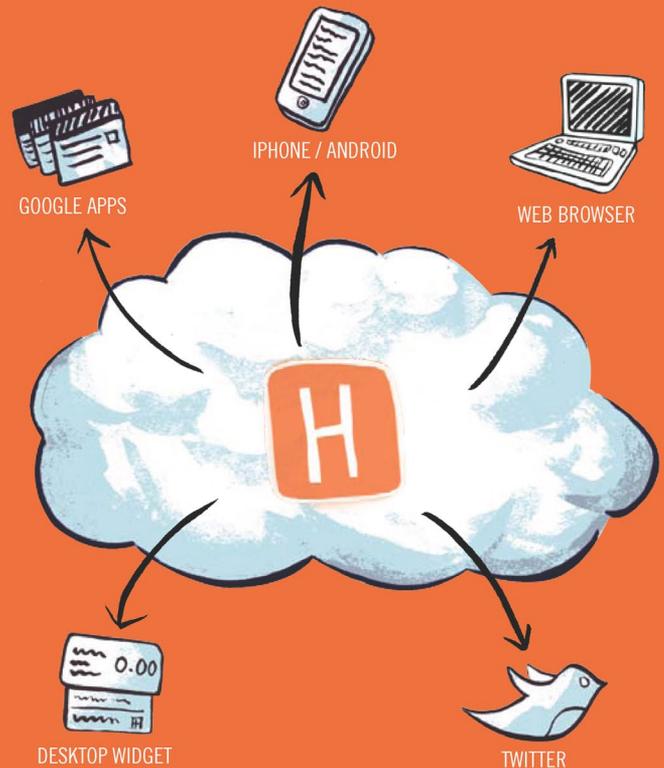
REPORTS

Track time anywhere, and invoice your clients with ease.

Harvest is available wherever your work takes you. Whether you are working from home, on-site, or through a flight. Harvest keeps a handle on your billable time so you can invoice accurately. Visit us and learn more about how Harvest can help you work better today.

Why Harvest?

- Convenient and accessible, anywhere you go.
- Get paid twice as fast when you send web invoices.
- Trusted by small businesses in over 100 countries.
- Ability to tailor to your needs with full API.
- Fast and friendly customer support.



Learn more at www.getHarvest.com/hackers



Dream. Design. Print.

MagCloud, the revolutionary new self-publishing web service by HP, is changing the way ideas, stories, and images find their way into peoples' hands in a printed magazine format.

HP MagCloud capitalizes on the digital revolution, creating a web-based marketplace where traditional media companies, upstart magazine publishers, students, photographers, designers, and businesses can affordably turn their targeted content into print and digital magazine formats.

Simply upload a PDF of your content, set your selling price, and HP MagCloud takes care of the rest—processing payments, printing magazines on demand, and shipping orders to locations around the world. All magazine formatted publications are printed to order using HP Indigo technology, so they not only look fantastic but there's no waste or overruns, reducing the impact on the environment.

Become part of the future of magazine publishing today at www.magcloud.com.

25% Off the First Issue You Publish

Enter promo code **HACKER** when you set your magazine price during the publishing process.

Please contact promo@magcloud.com with any questions.

MAGCLOUD