The most common resource bounds are based in time. The reason why is that time is by far the most expensive resource. Other resource bounds also yield time bounds frequently. For example, if an algorithm requires $\Omega(n^2)$ space, then it must require $\Omega(n^2)$ time to make use of that space.

Let's make precise what we mean by space complexity.

Let M be a deterministic TM that halts on all inputs. The <u>space complexity</u> of M is the function $f: \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of tape cells that M <u>scans</u> on any input of length n. If f is the space complexity of M, we also say that M runs in space $f(n)$.

The keyword scans above is important. If we used modified instead, M would be able to read all of its input on sublinear spaces. We will formalize a sublinear notion later when dealing with log spaces.

We also can define a notion of nondeterministic space complexity.

If M is a nondeterministic Turing Machine wherein all branches of computation halt on all inputs, we define its space complexity $f(n)$ to be the maximum number of tape cells modified on any branch of computation on any input of length n.

An important aspect of space complexity to note is that TMs are close enough to real computers to yield results that are immediately useful. One tape cell corresponds to one bit of memory.

Now just like time complexity, we can define space complexity classes.

Let $f: \mathbb{N} \to \mathbb{N}$ be a function. The <u>space complexity classes</u> SPACE(f(n)) and NSPACE(f(n)) are defined as follows.

$$\text{SPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n))\text{-space deterministic TM}\}$$

$$\text{NSPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n))\text{-space nondeterministic TM}\}$$

Let's get an exciting result about SAT using these definitions.

Ex)
```
M = "On input $\langle \phi \rangle$,        ($\phi$ is a Boolean formula)
    1) For each assignment of variables $x_1, \ldots, x_m$ of $\phi$
        a) Evaluate $\phi(x_1, \ldots, x_n)$
        b) Accept if $\phi$ evaluated to true
    2) Reject"
```

Notice that we can loop over assignments by simply counting up from 0^m by until we overflow back to 0^m. This can be done in place. Moreover, evaluating ϕ can be done by copying it and the substitution in the current assignment. Thus $SAT \in SPACE(n)$!

That's really good, right? We'll, yeah. For space. But the magic at work here is that space is reusable. This is similar to a nondeterministic TM "reusing" time. We can show $SAT \in NTIME(n)$ easily enough, but that doesn't tell us much on its own.

Now a reasonable question to ask is how SPACE and NSPACE relate to each other. We can actually answer this in one direction.

<u>Savitch's Thm</u> For any function $f: \mathbb{N} \to \mathbb{N}$, where $f(n) \in \Omega(\log n)$

$$NSPACE(f(n)) \subseteq SPACE(f(n)^2)$$

To prove this theorem, let's first consider a more elegant language with an interesting property.

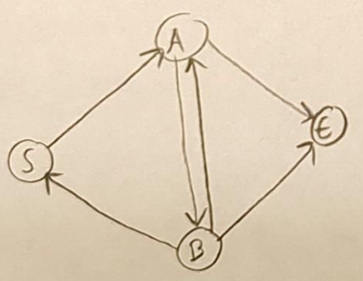
$$PATH = \{\langle G,s,t\rangle \mid G \text{ is a directed graph with an } s\text{-}t \text{ path}\}$$

We can write a nondeterministic algorithm to decide PATH.

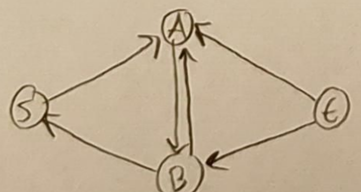
```
N = " On input <G,s,t>,
  1) Let v = s                     // v takes log V space
  2) For i = 1 to |V|              // i takes log V space
      a) If v = t, accept
      b) Pick a neighbor u of v
      c) u = v
  3) Reject"
```

A path from s to t requires at most |V| vertices, so if we happen to stumble upon it, we accept. If there is a path, then some branch of computation, however unlikely, will find it.

Ex)



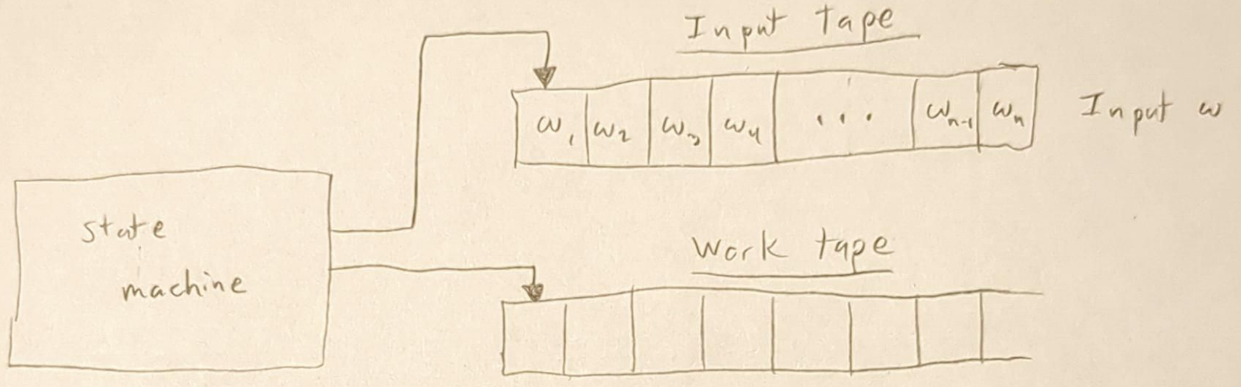
A/R	Possible paths	Result
A	SAt	Accept
R	SABS	
A	SABt	
R	SABA	



A/R	Possible paths	Result
R	SABt	Reject
R	SABA	

Notice that N only wrote to its tape to record v and j, both of which take only log V space to keep track of. We get that PATH ∈ NSPACE(log n).

In general, when working in sublinear spaces, we separate the input out onto a read-only tape and have a second work tape.



When we do this, we can define space complexities simply as the number of cells scanned on the work tape. The downside to this is to do work in place on the input, we have to copy it over first.

As an aside, we define

$$L = \text{SPACE}(\log n),$$
$$NL = \text{NSPACE}(\log n).$$

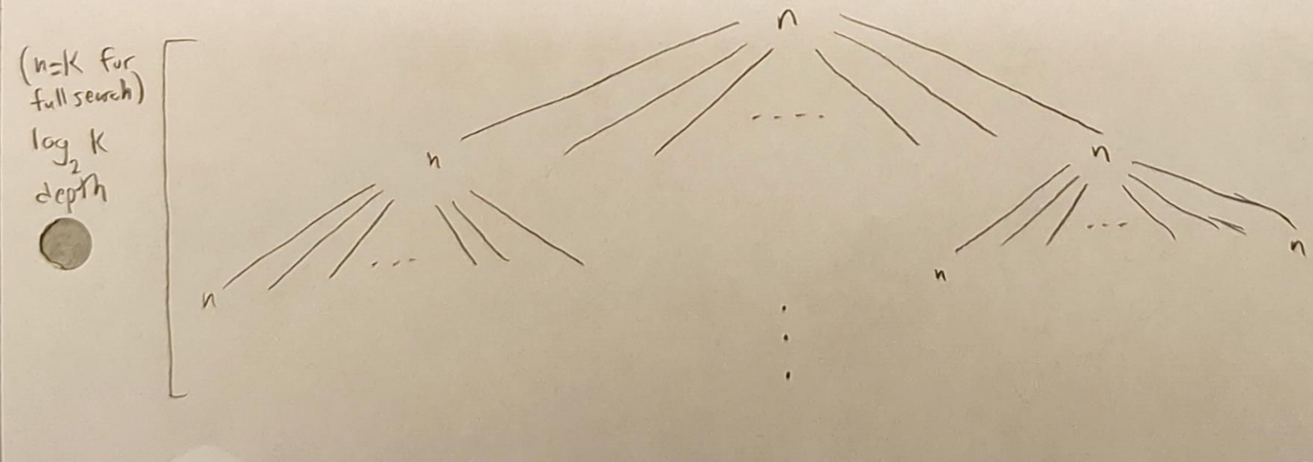
The log space algorithms have just enough working memory to do interesting things and have a number of interesting properties, which is why we're interested in L/NL and not, say, SPACE($n^{1/2}$) or NSPACE($\log^2 n$).

How do we turn N into a deterministic algorithm? There are dozens of path finding algorithms in P, but we want one that minimizes the required space. We use a technique known as divide and conquer to do this.

```
DPath(G, s, t, k)
    If $k = 0$
        Return $s = t$
    If $k = 1$
        Return $(s,t) \in E$
    for $u \in V$
        If $DPath(G, s, u, \lfloor \frac{k}{2} \rfloor) \wedge DPATH(G, u, t, \lceil \frac{k}{2} \rceil)$
            Return true/accept
    Return false/reject
```

This algorithm has recursive depth $\log V$ $(k \leq V)$, and each recursive call needs only track u and k, both of which only require $\log V$ space each. So DPath requires $O(2 \log V \log V) = O(\log^2 V)$. Thus DPath $\in SPACE(\log^2 n)$. That's fantastic... until one considers the runtime. $T(n,k)$.

$$T(n,k) = 2n\,T(n, \tfrac{k}{2}) + \Theta(n)$$



$S_0 = n$

$S_1 = 2n(n) = 2n^2$

$S_2 = 2n(2n^2) = 4n^3$

$\vdots$

$S_{i+1} = 2n S_i = 2^i n^{i+1}$

Let's total up the runtime of DPath to get an explicit formula for it.

$$\frac{1}{2}\sum_{i=0}^{\log n} S_i = n \sum_{i=0}^{\log_2 n} (2n)^i = n \frac{(2n)^{\log_2(n)+1} - 1}{2n-1} = n \frac{2n \cdot n^{\log_2 n} - 1}{2n - 1} \simeq n^{\log_2(n)+1}$$

So DPath has runtime $O(n^{\log_2(n)+1})$, which is very bad.

But for now, only space interests us, so this is fine.

Now why does this matter? Well N used $O(\log n)$ space, while DPath used $O(\log^2 n)$ space. This matches Savitch's theorem exactly (indeed, we can extend the theorem to functions $f \geq \log n$). Moreover, this is a general reachability result, so we should expect to get similar results about exploring configuration spaces to look for accepting configurations,

In N/DPath, we kept track of a vertex/vertex and recursive depth. If we generalize this to configurations requiring $f(n) \geq \log n$ space, then we keep track of a configuration/configuration and recursive depth. This requires $f(n) \cdot \lg(2^{\log_2 f(n)}) = f(n) \cdot f(n) = f(n)^2$ space, which yields the result we want, i.e. that $NSPACE(f(n)) \subseteq SPACE(f(n)^2)$. The first term here is how much space a configuration requires to store and the second term is the log of the maximum path length (or rather it is the recursive depth).