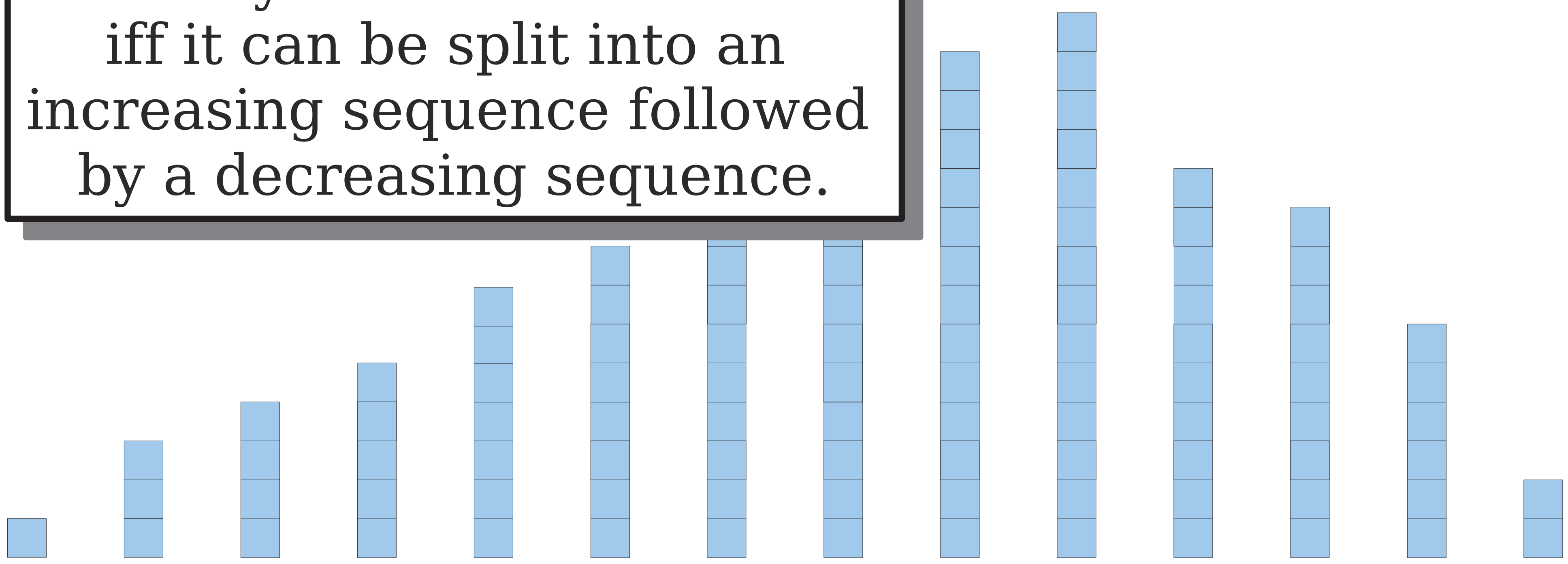
Another Algorithm:

Maximizing Unimodal Arrays

Unimodality

An array is called **unimodal** iff it can be split into an increasing sequence followed by a decreasing sequence.



1	3	4	5	7	8	10	12	13	14	10	9	6	2

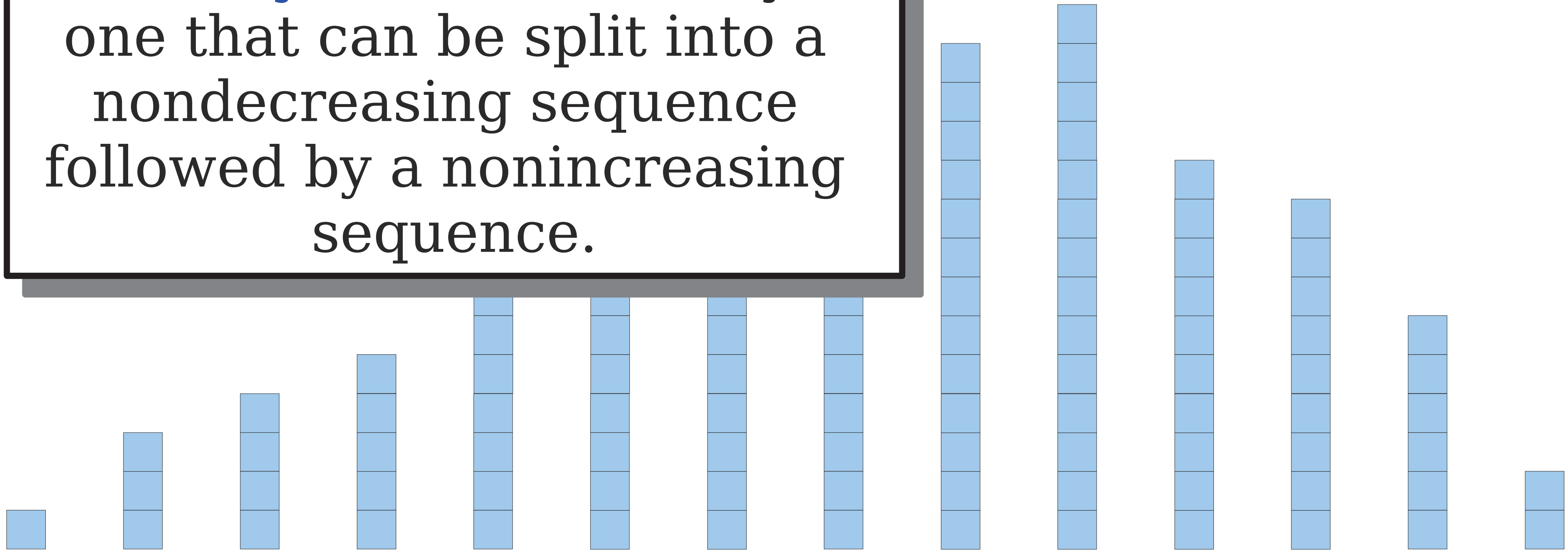
```
procedure unimodalMax(list A, int low, int high):
  if low = high - 1:
     return A[low]

  let mid = ⌊(high + low) / 2⌋
  if A[mid] < A[mid + 1]
     return unimodalMax(A, mid + 1, high)
  else:
     return unimodalMax(A, low, mid + 1)
```

$$T(1) = \Theta(1)$$
$$T(n) \leq T(\lceil n / 2 \rceil) + \Theta(1)$$

O(log *n*)

Unimodality II

A **weakly unimodal** array is one that can be split into a nondecreasing sequence followed by a nonincreasing sequence.



1	3	4	5	7	8	10	10	13	14	10	9	6	2

```
procedure weakUnimodalMax(list A, int low, int high):
  if low = high - 1:
     return A[low]

  let mid = ⌊(high + low) / 2⌋
  if A[mid] < A[mid + 1]
     return weakUnimodalMax(A, mid + 1, high)
  else if A[mid] > A[mid + 1]
     return weakUnimodalMax(A, low, mid + 1)
  else
     return max(weakUnimodalMax(A, low, mid + 1)
                weakUnimodalMax(A, mid + 1, high))
```

$$T(1) = \Theta(1)$$
$$T(n) \leq T(\lceil n / 2 \rceil) + T(\lfloor n / 2 \rfloor) + \Theta(1)$$

```
procedure weakUnimodalMax(list A, int low, int high):
  if low = high - 1:
     return A[low]

  let mid = ⌊(high + low) / 2⌋
  if A[mid] < A[mid + 1]
     return weakUnimodalMax(A, mid + 1, high)
  else if A[mid] > A[mid + 1]
     return weakUnimodalMax(A, low, mid + 1)
  else
     return max(weakUnimodalMax(A, low, mid + 1)
                weakUnimodalMax(A, mid + 1, high))
```

$$\mathrm{T}(1) \leq c$$
$$\mathrm{T}(n) \leq 2\mathrm{T}(n / 2) + c$$

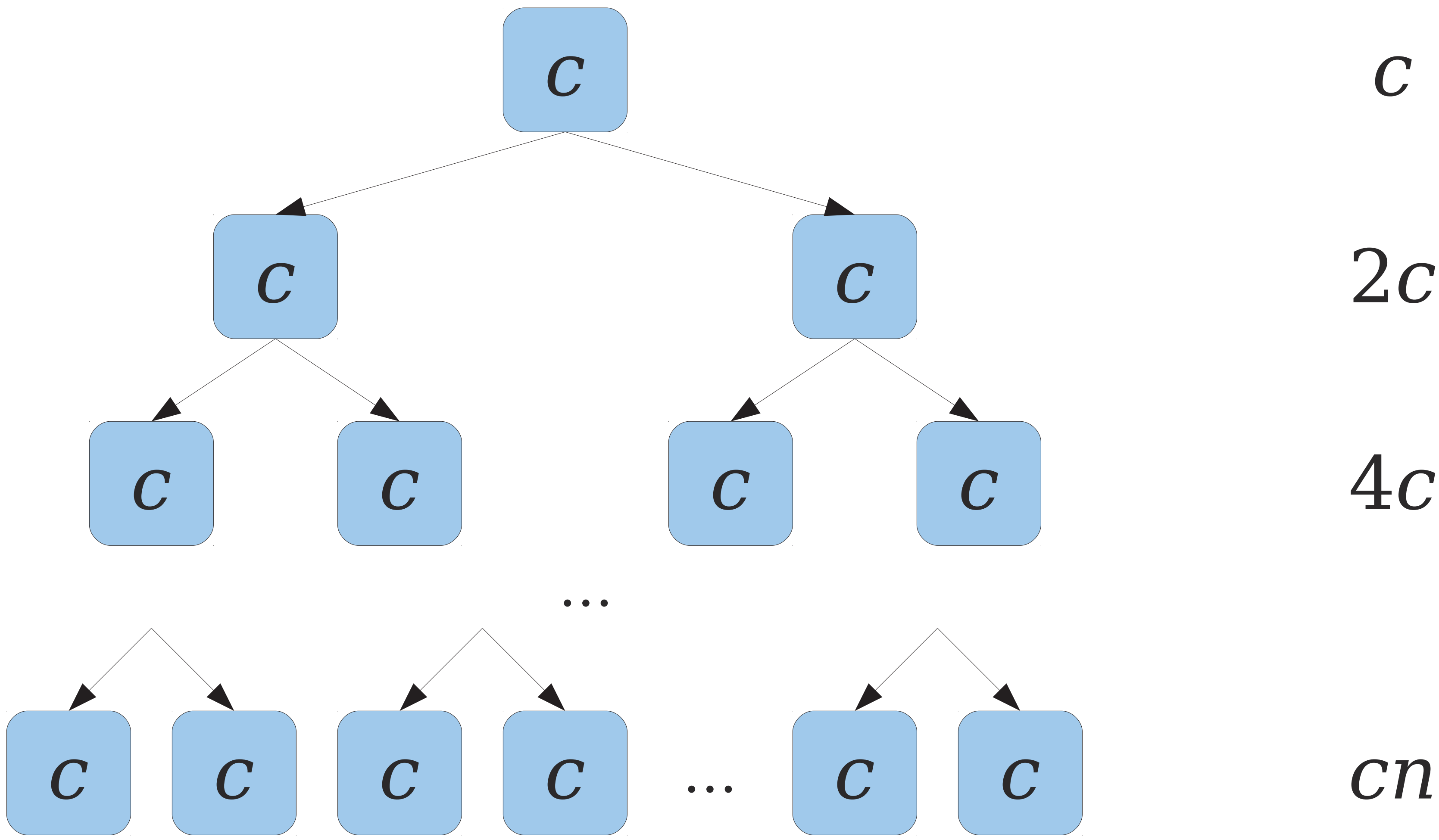
$$\boxed{\begin{aligned} &\mathrm{T}(1) \le c \\ &\mathrm{T}(n) \le 2\mathrm{T}(n/2) + c \end{aligned}}$$

$$\begin{aligned}
\mathrm{T}(n) &\le 2\mathrm{T}\left(\frac{n}{2}\right)+c \\
&\le 2\left(2\mathrm{T}\left(\frac{n}{4}\right)+c\right)+c \\
&\le 4\mathrm{T}\left(\frac{n}{4}\right)+2c+c \\
&= 4\mathrm{T}\left(\frac{n}{4}\right)+3c \\
&\le 4\left(2\mathrm{T}\left(\frac{n}{8}\right)+c\right)+3c \\
&= 8\mathrm{T}\left(\frac{n}{8}\right)+4c+3c \\
&= 8\mathrm{T}\left(\frac{n}{8}\right)+7c \\
&\cdots \\
&\le 2^k\mathrm{T}\left(\frac{n}{2^k}\right)+\left(2^k-1\right)c
\end{aligned}$$

$$\boxed{\begin{array}{l} \mathrm{T}(1) \le c \\ \mathrm{T}(n) \le 2\mathrm{T}(n / 2) + c \end{array}}$$

$$\begin{aligned}
\mathrm{T}(n) &\le 2^k\,\mathrm{T}\left(\frac{n}{2^k}\right) + (2^k - 1)c \\
&\le 2^{\log_2 n}\,\mathrm{T}(1) + (2^{\log_2 n} - 1)c \\
&= n\,\mathrm{T}(1) + c(n-1) \\
&\le cn + c(n-1) \\
&= 2cn - c \\
&= O(n)
\end{aligned}$$

$$T(1) \le c$$
$$T(n) \le 2T(n/2) + c$$



$$2cn - c$$

Another Recurrence Relation

- The recurrence relation

$$\boxed{\begin{aligned} &T(1) = \Theta(1) \\ &T(n) \leq T(\lceil n / 2 \rceil) + T(\lfloor n / 2 \rfloor) + \Theta(1) \end{aligned}}$$

  solves to $T(n) = \mathbf{O(n)}$

- Intuitively, the recursion tree is "bottomheavy:" the bottom of the tree accounts for almost all of the work.

Unimodal Arrays

- Our recurrence shows that the work done is $O(n)$, but this might not be a tight bound.
- Does our algorithm ever do $\Omega(n)$ work?
- **Yes:** What happens if all array values are equal to one another?
- Can we do better?

A Lower Bound

- **Claim**: Every correct algorithm for finding the maximum value in a unimodal array must do $\Omega(n)$ work in the worst-case.
- Note that this claim is over *all possible algorithms*, so the argument had better be watertight!

A Lower Bound

- We will prove that any algorithm for finding the maximum value of a unimodal array must, on at least one input, inspect all n locations.
- *Proof idea*: Suppose that the algorithm didn't do this.

0	0	0	0	0	0	?	0	0	0	0	0	0	0

A Lower Bound

- We will prove that any algorithm for finding the maximum value of a unimodal array must, on at least one input, inspect all n locations.
- *Proof idea*: Suppose that the algorithm didn't do this.

0	0	0	0	0	0	**10**	0	0	0	0	0	0	0

A Lower Bound

- We will prove that any algorithm for finding the maximum value of a unimodal array must, on at least one input, inspect all n locations.
- *Proof idea*: Suppose that the algorithm didn't do this.

0	0	0	0	0	0	**0**	0	0	0	0	0	0	0

Algorithmic Lower Bounds

- The argument we just saw is called an **adversarial argument** and is often used to establish algorithmic lower bounds.
- Idea: Show that if an algorithm doesn't do enough work, then it cannot distinguish two different inputs that require different outputs.
- Therefore, the algorithm cannot always be correct.

o Notation

- Let $f, g : \mathbb{N} \to \mathbb{N}$.
- We say that $\boldsymbol{f(n) = o(g(n))}$ (f is ***little-o*** of g) iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

- In other words, f grows strictly slower than g.
- Often used to describe impossibility results.
- For example: There is no $o(n)$-time algorithm for finding the maximum element of a weakly unimodal array.

What Does This Mean?

- In the worst-case, our algorithm must do $\Omega(n)$ work.
- That's the same as a linear scan over the input array!
- Is our algorithm even worth it?
- **Yes**: In most cases, the runtime is $\Theta(\log n)$ or close to it.