# Score-based Paragraph-level Line Breaking

kojii@
Apr 2023 - June 2023

## Overview

There are two basic approaches when breaking text into lines.

1. **Line-by-line**, also known as "**greedy**" algorithms.
2. **Paragraph-level** algorithms.

The **line-by-line algorithm** is described in more detail in the CSS `text-wrap: stable`. This type is fast, and is the default algorithm for all current browsers.

**Paragraph-level algorithm** takes other lines in the paragraph into consideration to achieve better typography. In CSS, the `text-wrap: pretty` is the property to opt-in to this type of the algorithm. This type generally produces better typography, with the cost of the performance. TeX is one of the most famous applications of this type.

Avoiding a short single word on the last line (typographic orphans) is one of the most visible advantages of the paragraph-level algorithm. They are often discouraged by web designers. When Blink shipped the `text-wrap: balance` property, many thought this was a solution for the typographic orphans (example articles 1, 2.) Though the balancing can minimize typographic orphans as an outcome, it makes the last line long to balance with other lines, which is good for headlines but not suitable for body text. The `text-wrap: pretty` is the property to minimize typographic orphans without such side effects.

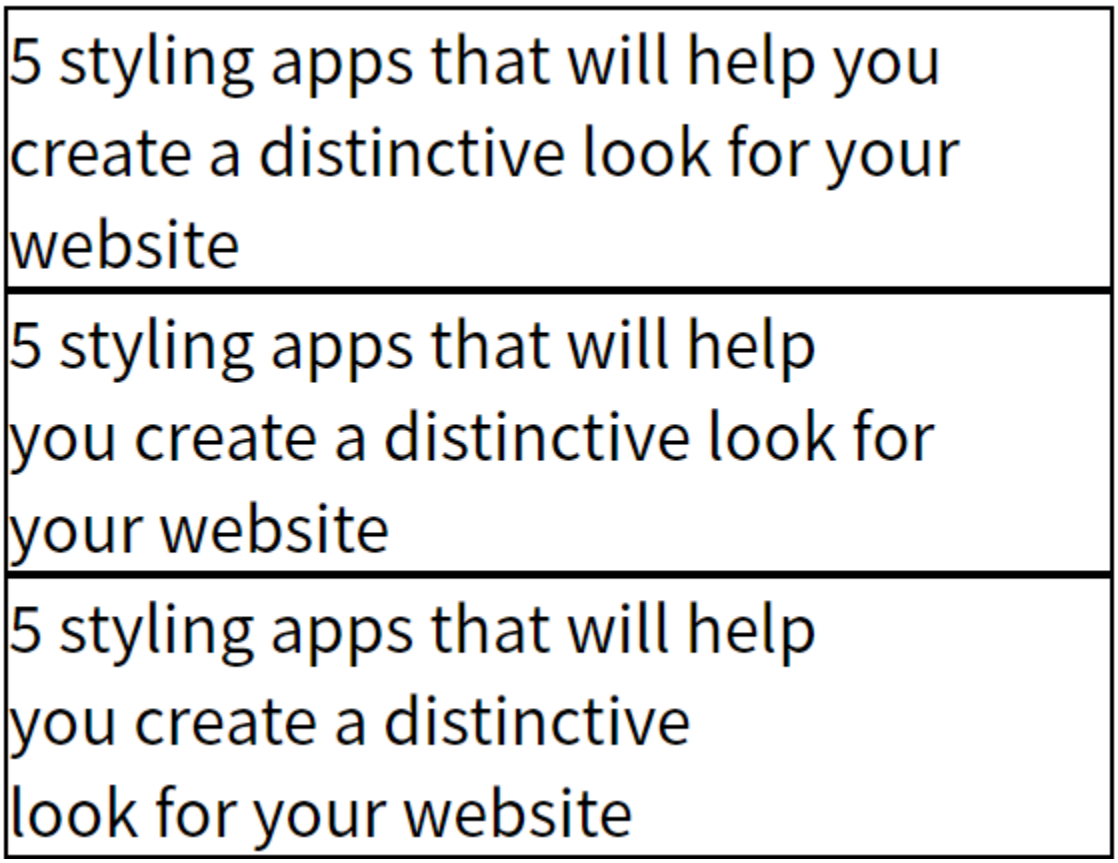
There are other possible advantages for paragraph-level line breaking, such as minimizing rivers. The csswg/#672 describes such other possible advantages. But the initial implementation focuses on typographic orphans, as it's the most visible benefit, and to minimize the performance impacts.

Because paragraph-level algorithms are slow, there are multiple variants to mitigate the performance impacts. Please see the Performance Considerations later in this document.

Tracking issue: crbug.com/1432798
Chromestatus: https://chromestatus.com/feature/5145771917180928

# `pretty` and `balance`

Both `pretty` and `balance` are paragraph-level line breaking, but they are designed for different purposes and produce different results.




These screenshots are taken from this video at Figma. The top example is by the line-by-line, greedy algorithm.
The second one is by `pretty`, which avoids typographic orphans.
The last one is by `balance`, which tries to make the lengths of all lines balanced as much as possible.
A developer feedback: I knew about the (new/cool) `text-wrap: balance;` — but sometimes that's a bit... too much. I feel like it's nice on headers but not smaller type.

# The Algorithm

The current implementation is based on the Knuth-Plass algorithm used in TeX, and on the Android's "optimal" line breaker, which is applied when BREAK_STRATEGY_HIGH_QUALITY is specified.

In short, it works as follows:

1. Compute all break candidates (a.k.a., break opportunities, the points where lines could break.)

2. Determine the penalty of each break candidate.
3. Compute the score for all possible combinations of break candidates.
4. The final break points are determined by the candidates with the best score.

If you are interested in more details of the Knuth-Plass algorithm, there are good public articles on the web, such as this.

The 2 and 3 are heuristic. In TeX, authors can adjust penalties and scores by macros, but CSS doesn't expose them as it would be difficult to make such controls interoperable. Good default values are desired.

Performance-wise, the 1 is quite expensive. LayoutNG improved the performance by minimizing the number of computations of break candidates compared to the legacy engine, but the score-based requires even more than the legacy. The 3 is also expensive, computing a score isn't too expensive but it iterates O(n!).

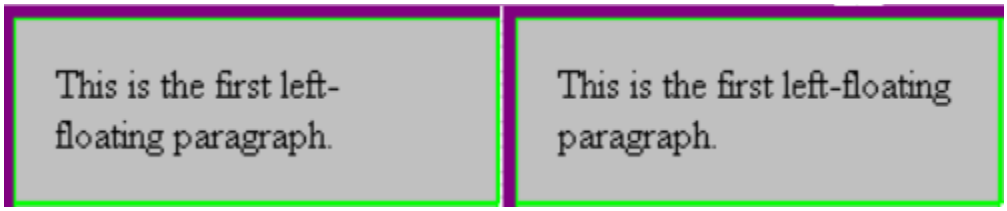
Also see the limitations below.

# Discussions/Cases

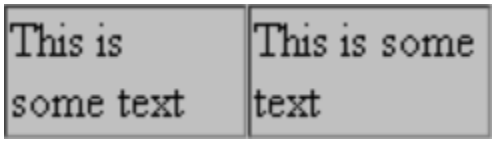
## Cases to Examine

Left: the score-based algorithm, right: the greedy algorithm.

- css1/formatting_model/floating_elements.html




- tables/mozilla/bugs/bug10009.html




## Needs Investigations

- [ ] compositing/overflow/do-not-paint-outline-into-composited-scrolling-contents.html
- [x] ~~css1/formatting_model/height_of_lines.html~~
- [x] ~~css1/units/color_units.html~~
- [ ] fast/block/float/float-on-line-obeys-container-padding.html
- [ ] fast/box-decoration-break/box-decoration-break-rendering.html
- [ ] fast/table/unbreakable-images-quirk.html
- [ ] paint/invalidation/overflow/overflow-hidden-to-visible.html

- [ ] virtual/text-antialias/ellipsis-in-justified-text.html
- [x] ~~virtual/text-antialias/hyphens/hyphens-auto-mock.html~~

### Discussions

- [css-text] Preventing too-short final lines of blocks (Last Line Minimum Length) #3473

# Performance Considerations

While the `text-wrap: pretty` property is an opt-in to accept slower line breaking, it shouldn't be too slow, or web developers can't use them due to their performance restrictions.

The pinpoint result when it is enabled for all web_tests is in this CL.

## Complexity

The score-based algorithm has different characteristics from the bisection algorithm. The bisection algorithm is O(*n* * log *w*) where *n* is the number of lines and *w* is the sum of spaces at the right end. The score-based algorithm is O(*n*! + *n*) where *n* is the number of break opportunities, so it will be slower if there are many break opportunities, such as when hyphenation is enabled.

Also, computing break opportunities are not cheap; it was one of LayoutNG's optimizations to minimize the number of computing break opportunities. The score-based algorithm will lose the benefit.

## Last 4 Lines

Because computing all break opportunities is expensive, and computing the score is O(n!) for the number of break opportunities, the number of break opportunities is critical for the performance. To minimize the performance impact, the implementation caches 4 lines ahead of the layout.

1. Before laying out a line, compute line breaking of 4 lines ahead of the layout.
2. If it finds the end of the block or a forced break, compute the score and optimize line breaks.
3. Otherwise layout the first line from the greedy line breaking results, and repeat this for the next line.

The line breaking results are cached, and used if the optimizer decided not to apply, to minimize the performance impact.

Currently, it applies to the last 4 lines of each paragraph, where "paragraph" is content in an inline formatting context separated by forced breaks.

## The Length of the Last Line

Because the benefit of the score-based line breaking is most visible when the last line of the paragraph is short, a performance optimization is to kick the optimizer in only when the last line is shorter than a ratio of the available width.

| | latin-ebook-resize | line-layout-line-height | japanese-kokoro |
|---|---|---|---|
| 1 (always) | +35% | +26% | +30% |
| 1/2 (ps#69) | +22% | +28% | +17% |
| 1/3 (ps#67) | +16% | +16% | +18% |
| 1/4 (ps#68) | +12% | +0.9% | +17% |
| 0 (never, ps#66) | +3.2% | -0.6% | -0.9% |

Currently, it applies only when the last line is equal to or less than ⅓ of the available width.

## Checking if the last line has only a single word

Checking if the last line has only a single word (i.e. no break opportunities) requires running the break iterator, but only once.

This optimization, in addition to the length of the last line to be shorter than ⅓, improves the pinpoint results (run 1, 2, 3, 4) by ~50%: latin-ebook-resize is +8.6%, line-layout-line-height +2%, japanese-kokoro +0.8%.

## Re-shaping Line-Start/-End

PS#29 experimented to see how much disabling re-shaping of line-start/-end can improve the performance. The result has improvements to the result without the experiment, but it doesn't look big enough when the experiment takes the risk of incorrect rendering of special fonts.

Currently, this optimization is not applied.

# Limitations

Following are the limitations as of ToT. The list may change in future. The "Disabled" means that the score-based algorithm is disabled for the condition.

| | **Score** | **Bisection** |
|---|---|---|
| Floats *1*2 | See *2 | Disabled |

| | | |
|---|---|---|
| Block fragmentation | Disabled | Disabled |
| Tabulation characters *1 | Disabled | Allowed |
| Forced breaks | Allowed | Disabled |
| Block-in-inline | Allowed | Disabled |
| Soft-hyphens *1 | Allowed | Allowed |
| Line overflow | Disabled | Disabled |
| CSS `box-decoration-break` *1 | Disabled | Allowed |
| CSS `break-all` | Allowed | Allowed |
| CSS `break-spaces` | Disabled | Allowed |
| CSS `column-span: all` | Disabled | Disabled |
| CSS `::first-line` *1 | Disabled | Allowed |
| CSS hyphens *1 | Allowed | Allowed |
| CSS initial-letter *1 | Disabled | Disabled |
| CSS line-clamp | Allowed | Allowed |
| CSS negative margins | Disabled | Allowed |
| CSS `overflow-wrap` *1 | See *3 | Allowed |
| CSS `text-indent` | Allowed | Allowed |

## 1. When Styles Change by Different Line Break Points

It is generally challenging for paragraph-level line breaking algorithms if styles change by different line break points. It is less challenging for the bisection algorithm, because it runs the actual line breaker for every candidate.

It’s technically possible to support them, just that they require additional logic, and the complexities vary by features. Some such features are supported, such as kernings or hyphenations, but not all features yet.

## 2. Floating Objects

Floating objects are one of cases explained in 1, and technically it is even more challenging than other features because it needs to know line heights, which is available only after layout.

But the demand to support floats turned out to be high (crbug.com/1440456), one of the highest requests among the current limitations.

Given that, blocks with floating objects are supported when it is “simple.” A “simple” block is when all following conditions are met:

- It has only one floating object, or multiple floating objects that create a rectangular exclusion.
- All floating objects must be at the beginning of the block.
- All lines must be known to have the same line height before the layout. This means:
  - All used fonts must be equal to or shorter than the block’s first available font.
  - It doesn’t have atomic inlines and other objects that need to be laid out to compute height.
  - It doesn’t use the `vertical-align` property.

An example of the “simple” block with floats:

```
<div style=”text-wrap: pretty”>
  <div style=”float: left; width: 100px; height: 20px”>...</div>
  Text text text…
</div>
```

Under these conditions, the current implementation can compute line heights before the layout.

### 3. CSS `overflow-wrap`

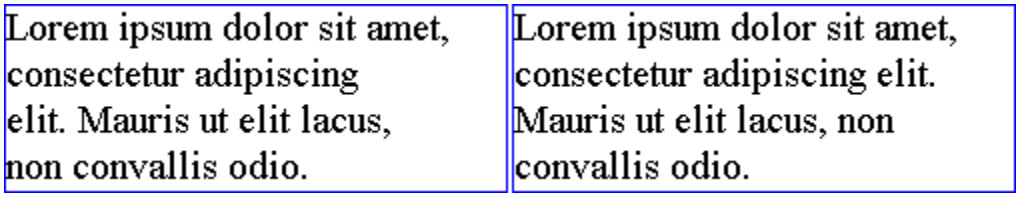
Applying the property doesn’t disable the algorithm, but when there are words that actually overflow the line and the fallback behavior is kicked in, it disables, for the same reason as 1.

## Balancing by the Score-based Algorithm

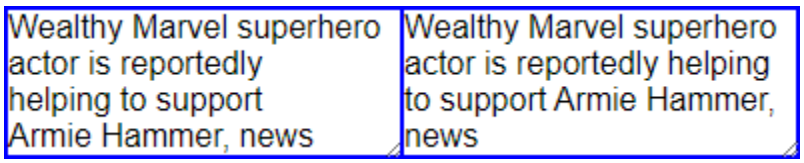
By adjusting penalties, the score-based algorithm can be used to balance lines. The results are similar to the bisection algorithm, but sometimes it produces better results.

Examples of the two algorithms for the same text:

Left: score, right: bisection, from text-wrap-balance-layout.html




Left: score, right: bisection, from www.yahoo.com (edited to fit into 4 lines)



The difference in the 2nd example comes from the fact that the bisection applies the same available width to all lines. In this example, if the box is slightly narrower, the last words of all 3 lines can't fit and it increases the number of lines. The score-based can handle such a situation by not making the first line narrower.

The two algorithms have different performance characteristics. The score-based is ordered by the number of break opportunities, while bisection is ordered by the number of lines and the sum of spaces on the right end. The comparison against the bisection line breaker is here.

Also they have different limitations. They can complement each other for cases where the other algorithm doesn't support.

This topic is tracked in a separate issue: crbug.com/1451205.

# Links

- Web developer feedback
  - One of 3 moments in our talk at Figma that got clapped for by the audience.
  - This Frontend reddit talks about "balance is a game changer," but wanting an option for body or anything other than headlines.
  - https://github.com/w3c/csswg-drafts/issues/3473#issuecomment-1620737159
    - https://stackoverflow.com/questions/31974448/how-can-i-prevent-having-just-one-hanging-word-on-a-new-line-in-an-html-element (upvoted 42 times)
    - https://stackoverflow.com/questions/4823722/how-can-i-avoid-one-word-on-the-last-line-with-css (upvoted 19 times)
    - https://stackoverflow.com/questions/38296454/prevent-line-break-between-last-word-of-an-element-and-another-element (upvoted 6 times)
  - Text Wrap Pretty is coming to CSS
  - In M117 Canary balanced + hyphens now look like this
  - text-wrap Rendering Performance
- Gecko
  - Intent to ship: CSS `text-wrap: balance`
- WebKit
  - Provide an initial implementation of text-wrap: balance
- References
  - Knuth, D., Plass, M. Breaking paragraphs into lines. Software Practice and Experience, 1981.