Strategies for Stable Merge Sorting

Authors:
Sam Buss, Alexander Knop

Institute:
U.C. San Diego

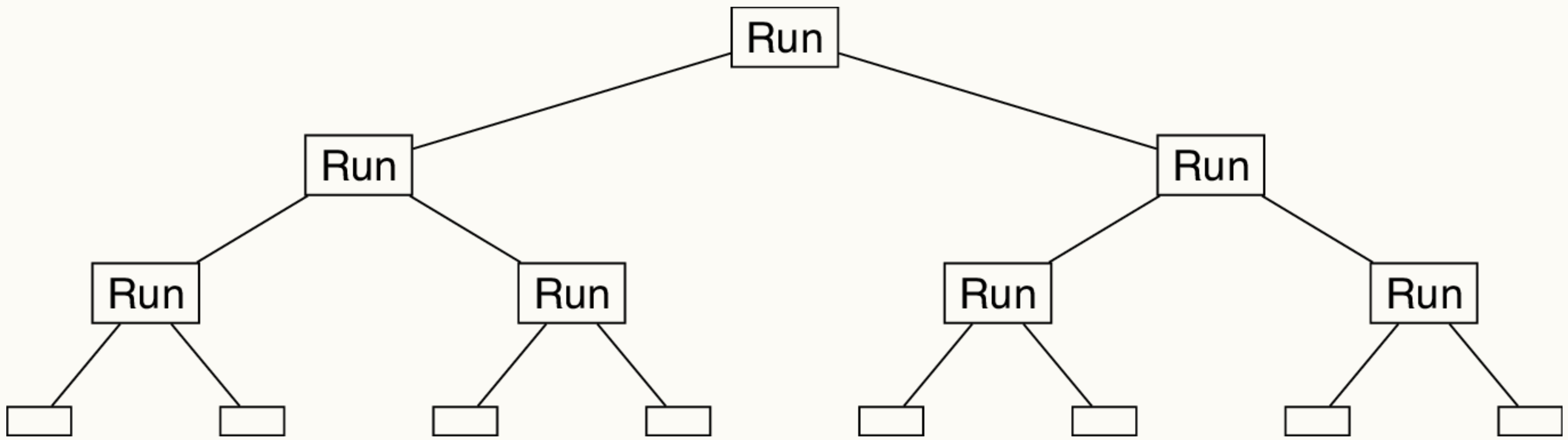
Sorting problem

It is impossible to sort with runtime less than $n \log n$.

Sorting problem

It is impossible to sort with runtime less than $n \log n$. **Merge Sort** also known as "von Neumann sort" is one of the first sorting algorithms with $O(n \log n)$ runtime.

Basic Von Neumann merge sort



- A "run" is an ascending sequence.
- Input consists of runs of size 1 (at leaves).
- Output: a single run containing all inputs (at the root).
- Formed by binary tree of merges combining runs.
- Runtime is $O(n \log n)$, where $n =$ input size.

Merge sort is readily made stable, by merging only adjacent runs.

Bottom up algorithm for Von Neumann Sort

```
def von_neumann_sort(S, n)
    Q = [] # Stack of runs
    while S.empty? do
      Q.push(Run.new(S.pop, 1)), l = Q.size
      while Q[l - 2].size < 2 * Q[l - 1].size do
        Q.merge(l - 2, l - 1)
      end
    end
    Q.merge(Q.size - 2, Q.size - 1) while Q.size > 1
    return Q[0]
end
```

Sorting problem

It is impossible to sort with runtime less than $n \log n$. **Merge Sort** also known as "von Neumann sort" is one of the first sorting algorithms with $O(n \log n)$ runtime.

Sorting problem

It is impossible to sort with runtime less than $n \log n$. **Merge Sort** also known as "von Neumann sort" is one of the first sorting algorithms with $O(n \log n)$ runtime.

- If we know something about the elements of the list we can sort much faster e.g. if all the elements of the list are integers with c digits, then the running time of the Digit Sort is $O(n \log c)$.

Sorting problem

It is impossible to sort with runtime less than $n \log n$. **Merge Sort** also known as "von Neumann sort" is one of the first sorting algorithms with $O(n \log n)$ runtime.

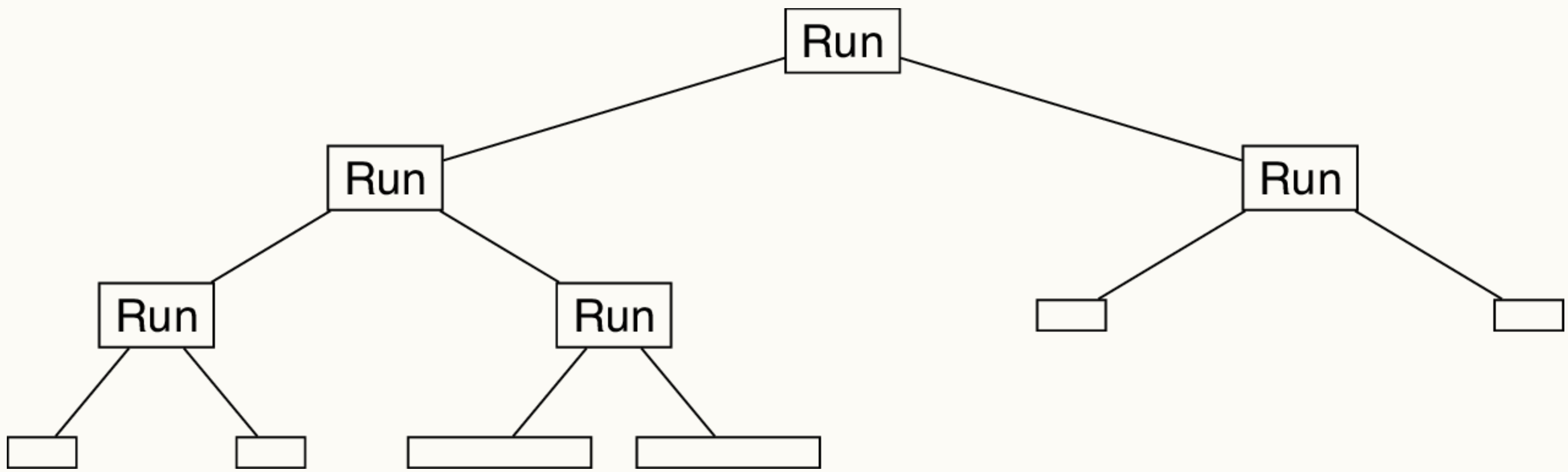
- If we know something about the elements of the list we can sort much faster e.g. if all the elements of the list are integers with c digits, then the running time of the Digit Sort is $O(n \log c)$.
- If the array is partially presorted we again can sort much faster e.g. if it is possible to split the list into m sorted subsequences (called "runs"), then the running time of the **Natural Merge Sort** (suggested by Knuth) is $O(n \log m)$.

Sorting problem

It is impossible to sort with runtime less than $n \log n$. **Merge Sort** also known as "von Neumann sort" is one of the first sorting algorithms with $O(n \log n)$ runtime.

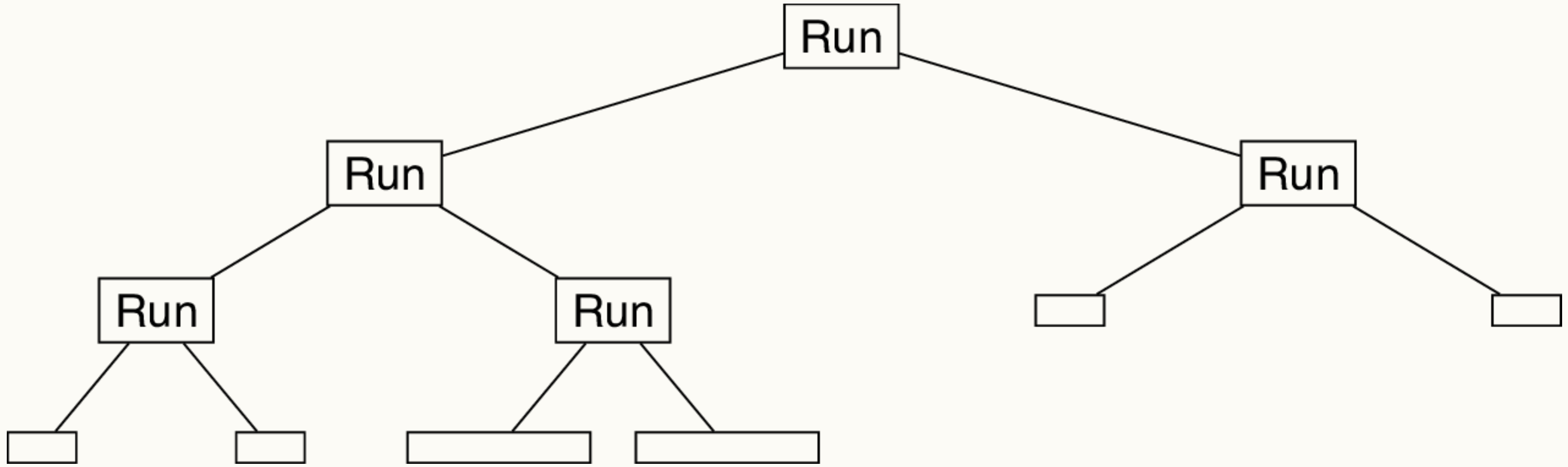
- If we know something about the elements of the list we can sort much faster e.g. if all the elements of the list are integers with c digits, then the running time of the Digit Sort is $O(n \log c)$.
- If the array is partially presorted we again can sort much faster e.g. if it is possible to split the list into m sorted subsequences (called "runs"), then the running time of the **Natural Merge Sort** (suggested by Knuth) is $O(n \log m)$. Natural Merge Sort identify runs which are already represent in the input.

Unequal run sizes - left-to-right binary merging



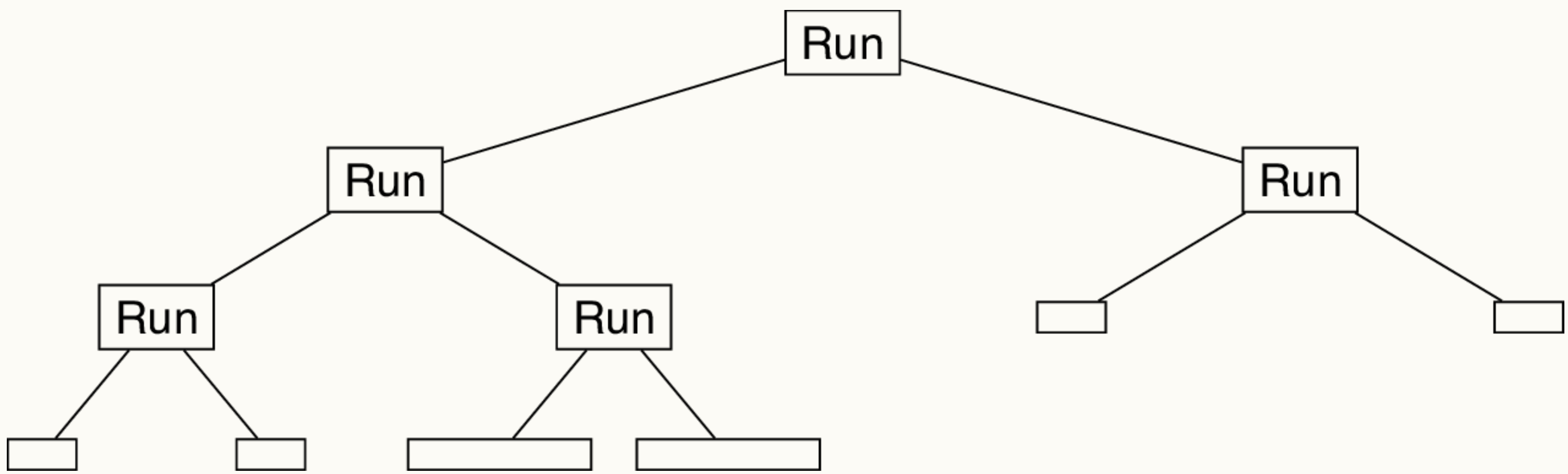
- Merging follows same left-to-right binary tree pattern as before.

Unequal run sizes - left-to-right binary merging



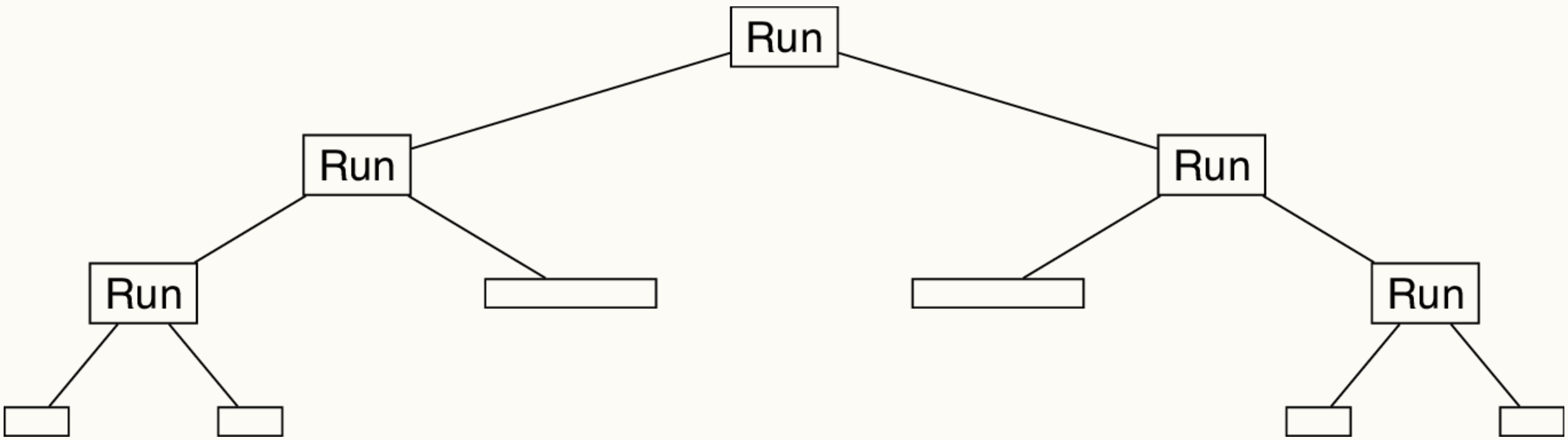
- Merging follows same left-to-right binary tree pattern as before.
- Inefficiency: the two longer runs are merged too soon.

Unequal run sizes - left-to-right binary merging

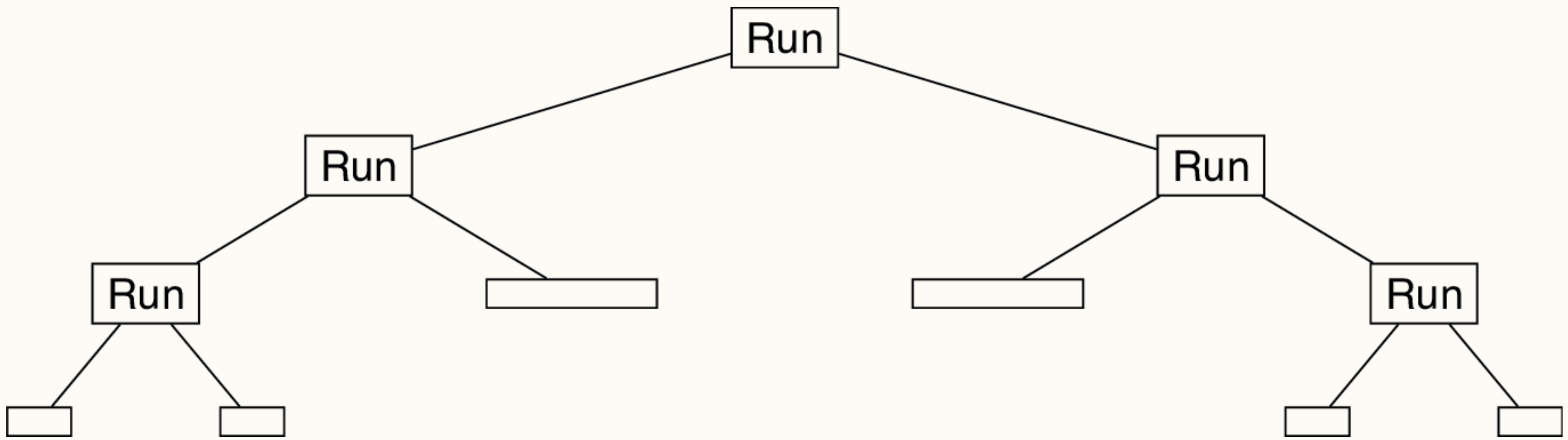


- Merging follows same left-to-right binary tree pattern as before.
- Inefficiency: the two longer runs are merged too soon. More efficient to delay merging them...

Unequal run sizes - more efficient merging

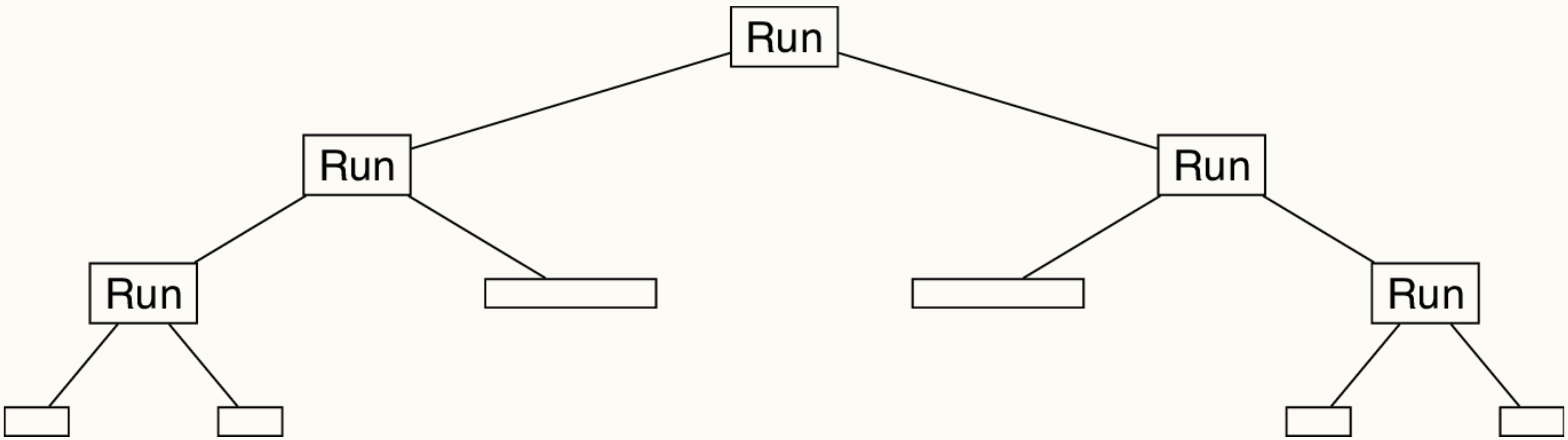


Unequal run sizes - more efficient merging



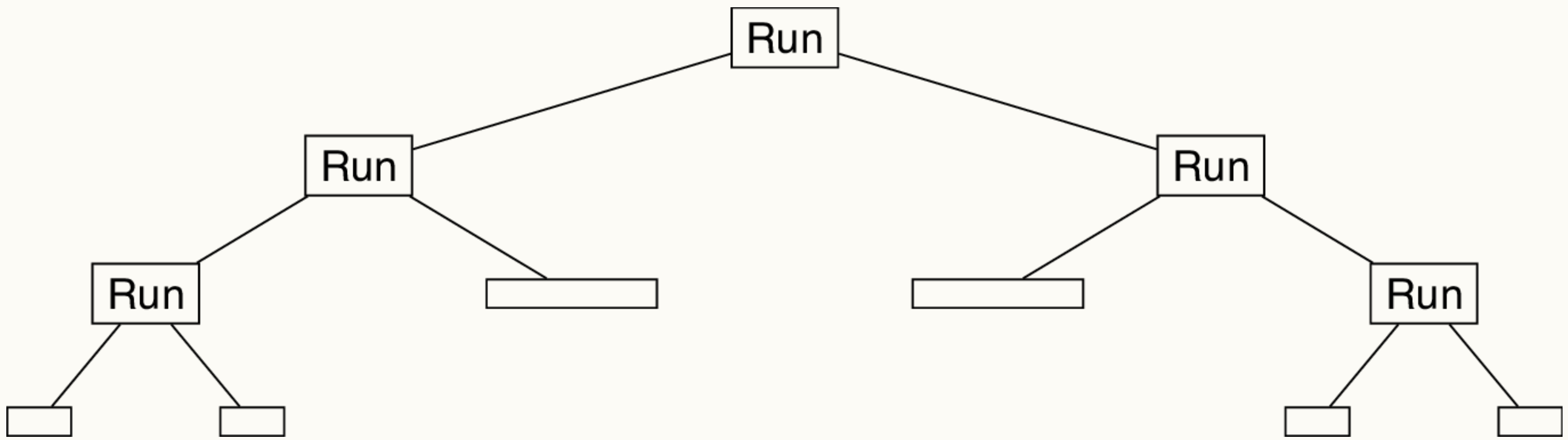
- **Merge Cost:** To merge runs of size ℓ_1, ℓ_2 takes time $\Theta(\ell_1 + \ell_2)$.

Unequal run sizes - more efficient merging



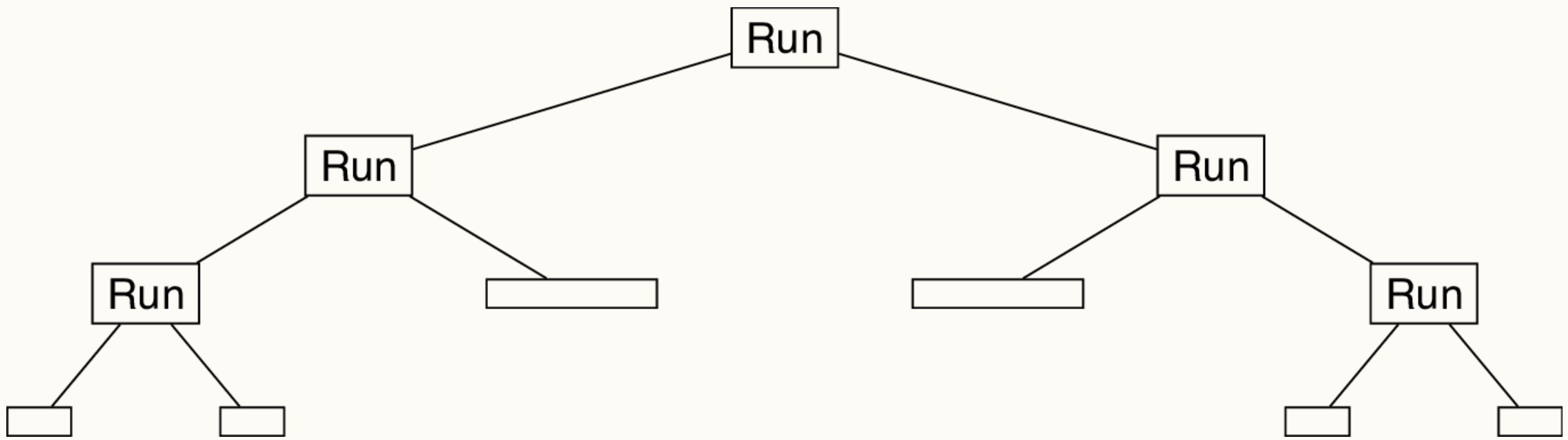
- **Merge Cost:** To merge runs of size ℓ_1, ℓ_2 takes time $\Theta(\ell_1 + \ell_2)$. We measure runtime by summing merge cost of all merges.

Unequal run sizes - more efficient merging



- **Merge Cost:** To merge runs of size ℓ_1, ℓ_2 takes time $\Theta(\ell_1 + \ell_2)$. We measure runtime by summing merge cost of all merges.
- Merge cost upper bounds comparison cost, and essentially matches comparison cost in most implementations.

Unequal run sizes - more efficient merging



- **Merge Cost:** To merge runs of size ℓ_1, ℓ_2 takes time $\Theta(\ell_1 + \ell_2)$. We measure runtime by summing merge cost of all merges.
- Merge cost upper bounds comparison cost, and essentially matches comparison cost in most implementations.

An **adaptive** merge sort chooses the order of merges to minimize the merge cost.

Basic framework for merge sorts: (k_1, k_2)-aware

```
def generic_merge_sort(S, n)
    Q = []
    while Q.size > 1 or not S.empty? do
        l = Q.size
        if merge?(Q[l - k_1].size,
                  Q[l - k_1 + 1].size,
                  ...,
                  Q[l - 1].size,
                  S.empty?) then
            i = choose_runs(Q) # l - k_2 <= i < l - 1
            Q.merge(i, i + 1)
        else
            Q.push(S.pop_run())
        end
    end
    return Q[0]
end
```

TimSort

TimSort is a natural merge sort - defined on next slide.

TimSort

TimSort is a natural merge sort - defined on next slide.

- TimSort is perhaps the most widely deployed sort ever, used in Python, Android, Java, and some browsers.

TimSort

TimSort is a natural merge sort - defined on next slide.

- TimSort is perhaps the most widely deployed sort ever, used in Python, Android, Java, and some browsers.
- Due to Tim Peters [*Python-Dev* blog post, 2002].

TimSort

TimSort is a natural merge sort - defined on next slide.

- TimSort is perhaps the most widely deployed sort ever, used in Python, Android, Java, and some browsers.
- Due to Tim Peters [*Python-Dev* blog post, 2002].
- TimSort is 4-aware; indeed (4,3)-aware. Based on the top four runs' sizes, chooses whether to merge some pair of runs.

TimSort

TimSort is a natural merge sort - defined on next slide.

- TimSort is perhaps the most widely deployed sort ever, used in Python, Android, Java, and some browsers.
- Due to Tim Peters [*Python-Dev* blog post, 2002].
- TimSort is 4-aware; indeed (4,3)-aware. Based on the top four runs' sizes, chooses whether to merge some pair of runs.
- Designed to work well both with partially sorted data, and with $n \log n$ worst-case runtime.

TimSort

TimSort is a natural merge sort - defined on next slide.

- TimSort is perhaps the most widely deployed sort ever, used in Python, Android, Java, and some browsers.
- Due to Tim Peters [*Python-Dev* blog post, 2002].
- TimSort is 4-aware; indeed (4,3)-aware. Based on the top four runs' sizes, chooses whether to merge some pair of runs.
- Designed to work well both with partially sorted data, and with $n \log n$ worst-case runtime.
- Has received little academic study until recently.

TimSort

```
def tim_sort(S, n)
  Q = []
  while S.empty? do
    Q.push(S.pop_run()), l = Q.size
    while true do
      if Q[l - 3].size < Q[l - 1].size then
        Q.merge(l - 3, l - 2)
      elsif  Q[l - 3].size <=
                 Q[l - 2].size + Q[l - 1].size
          or Q[l - 4].size <=
                 Q[l - 3].size + Q[l - 2].size
          or Q[l - 2].size <= Q[l - 1].size then
        Q.merge(l - 2, l - 1)
      else
        break
      end
    end
  end
  Q.merge(Q.size - 2, Q.size - 1) while Q.size > 1
  return Q[0]
end
```

TimSort

Intuition for TimSort:

```
Q[i].size > Q[i + 1].size + Q[i + 2].size
```

is maintained.

TimSort

Intuition for TimSort:

```
Q[i].size > Q[i + 1].size + Q[i + 2].size
```

is maintained. Thus stack `Q` has logarithmic height; and runtime is $O(n \log n)$.

TimSort

Intuition for TimSort:

```
Q[i].size > Q[i + 1].size + Q[i + 2].size
```

is maintained. Thus stack `Q` has logarithmic height; and runtime is $O(n \log n)$. This is true of the corrected version of [dGRBBH]; they corrected this while developing a formal proof of correctness.

TimSort

Intuition for TimSort:

```
Q[i].size > Q[i + 1].size + Q[i + 2].size
```

is maintained. Thus stack `Q` has logarithmic height; and runtime is $O(n \log n)$. This is true of the corrected version of [dGRBBH]; they corrected this while developing a formal proof of correctness.

THEOREM (AJNP'18)

TimSort has runtime $O(n \log m)$. (Proving a conjecture of [Buss-K.'19]).

TimSort

Intuition for TimSort:

```
Q[i].size > Q[i + 1].size + Q[i + 2].size
```

is maintained. Thus stack `Q` has logarithmic height; and runtime is $O(n \log n)$. This is true of the corrected version of [dGRBBH]; they corrected this while developing a formal proof of correctness.

THEOREM (AJNP'18)

TimSort has runtime $O(n \log m)$. (Proving a conjecture of [Buss-K.'19]).

THEOREM (BUSS-K.'19)

TimSort has worst-case merge cost $\geq (1.5 - o(1))n \log n$.

Summary of merge costs upper/lower bounds

Algorithm	Upper bound	Lower bound
TimSort	$O(n \log m)$	$1.5 \cdot n \log n$ [Buss-K.'19]
α-stack sort	$O(n \log n)$ [ANP'15]	$c_\alpha \cdot n \log n$ $\omega(n \log m)$ [Buss-K.'19]
Shivers sort	$n \log n$ [Shivers'02]	$\omega(n \log m)$ [Buss-K.'19]
2-merge sort	$c_2 \cdot n \log m$ [Buss-K.'19]	$c_2 \cdot n \log n$ [Buss-K.'19]
α-merge sort	$c_\alpha \cdot n \log m$ [Buss-K.'19]	$c_\alpha \cdot n \log n$ [Buss-K.'19]

for $\varphi < \alpha \leq 2$. φ is the golden ratio. Bounds are asymptotic.

Summary of merge costs upper/lower bounds

Algorithm	Upper bound	Lower bound
TimSort	$O(n \log m)$	$1.5 \cdot n \log n$ [Buss-K.'19]
α-stack sort	$O(n \log n)$ [ANP'15]	$c_\alpha \cdot n \log n$ $\omega(n \log m)$ [Buss-K.'19]
Shivers sort	$n \log n$ [Shivers'02]	$\omega(n \log m)$ [Buss-K.'19]
2-merge sort	$c_2 \cdot n \log m$ [Buss-K.'19]	$c_2 \cdot n \log n$ [Buss-K.'19]
α-merge sort	$c_\alpha \cdot n \log m$ [Buss-K.'19]	$c_\alpha \cdot n \log n$ [Buss-K.'19]

for $\varphi < \alpha \leq 2$. φ is the golden ratio. Bounds are asymptotic.
The constants c_2 and c_α satisfy:

$$c_2 = 3/\log(27/4) \approx 1.08897.$$

$$1.042 < c_\alpha \leq c_2$$

for $\varphi < \alpha \leq 2$.

2-Stack Sort

The 2-stack sort can be viewed similar to a "naturalized, adaptive" von Neumann sort.

```
def two_stack_sort(S, n)
  Q = []
  while S.empty? do
    Q.push(S.pop_run()), l = Q.size
    while Q[l - 2].size < 2 * Q[l - 1].size do
      Q.merge(l - 2, l - 1)
    end
  end
  Q.merge(Q.size - 2, Q.size - 1) while Q.size > 1
  return Q[0]
end
```

2-Merge Sort - Intuition

2-merge sort merges either `Q[l - 3]` and `Q[l - 2]` or merges `Q[l - 2]` and `Q[l - 3]`.
Target invariant: Maintain

```
Q[0].size >= 2 * Q[1].size >=
                 4 * Q[2].size >= ...
```

2-Merge Sort - Intuition

2-merge sort merges either `Q[l - 3]` and `Q[l - 2]` or merges `Q[l - 2]` and `Q[l - 3]`.
Target invariant: Maintain

```
 Q[0].size >= 2 * Q[1].size >=
                4 * Q[2].size >= ...
```

Whenever invariant is violated: it will be violated by the top two elements `Q[l - 2]` and `Q[l - 1]`. When this happens, merge `Q[l - 2]` with the smaller of `Q[l - 3]` and `Q[l - 1]`.

2-Merge Sort

```
def two_merge_sort(S, n)
  Q = []
  while S.empty? do
    Q.push(S.pop_run()), l = Q.size
    while Q[l - 2].size < 2 * Q[l - 1].size do
      if Q[l - 3].size < Q[l - 1].size then
        Q.merge(l - 3, l - 2)
      else
        Q.merge(l - 2, l - 1)
    end
  end
  Q.merge(Q.size - 2, Q.size - 1) while Q.size > 1
  return Q[0]
end
```

α-Merge Sort ($\varphi < \alpha \leq 2$)

```
def alpha_merge_sort(S, n, alpha)
  Q = []
  while S.empty? do
    Q.push(S.pop_run()), l = Q.size
    while Q[l - 2].size < alpha * Q[l - 1].size
        and Q[l - 3].size < alpha * Q[l - 2].size do
      if Q[l - 3].size < Q[l - 1].size then
        Q.merge(l - 3, l - 2)
      else
        Q.merge(l - 2, l - 1)
    end
  end
  Q.merge(Q.size - 2, Q.size - 1) while Q.size > 1
  return Q[0]
end
```

Lower/Upper bounds for 2-Merge and α-Merge Sorts

Define $c_2 = 3/\log(27/4) \approx 1.08897$.

Define $c_\alpha = \dfrac{\alpha+1}{(\alpha+1)\log(\alpha+1) - \alpha\log\alpha}$.

THEOREM (BUSS-K.'19)

① The worst case merge-cost of 2-merge sort is $(c_2 - o(1))n\log n$.

Lower/Upper bounds for 2-Merge and α-Merge Sorts

Define $c_2 = 3/\log(27/4) \approx 1.08897$.

Define $c_\alpha = \dfrac{\alpha+1}{(\alpha+1)\log(\alpha+1) - \alpha\log\alpha}$.

THEOREM (BUSS-K.'19)

1. The worst case merge-cost of 2-merge sort is $(c_2 - o(1))n\log n$.
2. The worst case merge-cost of α-merge sort is $(c_\alpha - o(1))n\log n$.

Lower/Upper bounds for 2-Merge and α-Merge Sorts

Define $c_2 = 3/\log(27/4) \approx 1.08897$.

Define $c_\alpha = \dfrac{\alpha+1}{(\alpha+1)\log(\alpha+1) - \alpha\log\alpha}$.

THEOREM (BUSS-K.'19)

1. The worst case merge-cost of 2-merge sort is $(c_2 - o(1))n\log n$.
2. The worst case merge-cost of α-merge sort is $(c_\alpha - o(1))n\log n$.
3. The 2-merge sort has merge-cost $\leq (c_2 + o(1))n\log m$.

Lower/Upper bounds for 2-Merge and α-Merge Sorts

Define $c_2 = 3/\log(27/4) \approx 1.08897$.

Define $c_\alpha = \dfrac{\alpha+1}{(\alpha+1)\log(\alpha+1) - \alpha\log\alpha}$.

THEOREM (BUSS-K.'19)

1. The worst case merge-cost of 2-merge sort is $(c_2 - o(1))n\log n$.
2. The worst case merge-cost of α-merge sort is $(c_\alpha - o(1))n\log n$.
3. The 2-merge sort has merge-cost $\leq (c_2 + o(1))n\log m$.
4. The α-merge sort has merge-cost $\leq (c_\alpha + o(1))n\log m$.

Subsequent work

THEOREM (MW'18)

There is a stable merge sorting algorithm similar to 3-aware algorithms achieving upper bounds and worst-case lower bounds equal to $n \log m$.

Subsequent work

THEOREM (MW'18)

There is a stable merge sorting algorithm similar to 3-aware algorithms achieving upper bounds and worst-case lower bounds equal to $n \log m$.

THEOREM (JUGE'18)

There is 3-aware algorithm achieving upper bounds and worst-case lower bounds equal to $n \log m$.

Subsequent work

THEOREM (MW'18)

There is a stable merge sorting algorithm similar to 3-aware algorithms achieving upper bounds and worst-case lower bounds equal to $n \log m$.

THEOREM (JUGE'18)

There is 3-aware algorithm achieving upper bounds and worst-case lower bounds equal to $n \log m$.

Moreover, the upper bounds have the form $(1 + o(1))n\mathcal{H}$, where $\mathcal{H}$ is the entropy-based *optimum, non-stable* merge-cost.

Subsequent work

THEOREM (MW'18)

There is a stable merge sorting algorithm similar to 3-aware algorithms achieving upper bounds and worst-case lower bounds equal to $n \log m$.

THEOREM (JUGE'18)

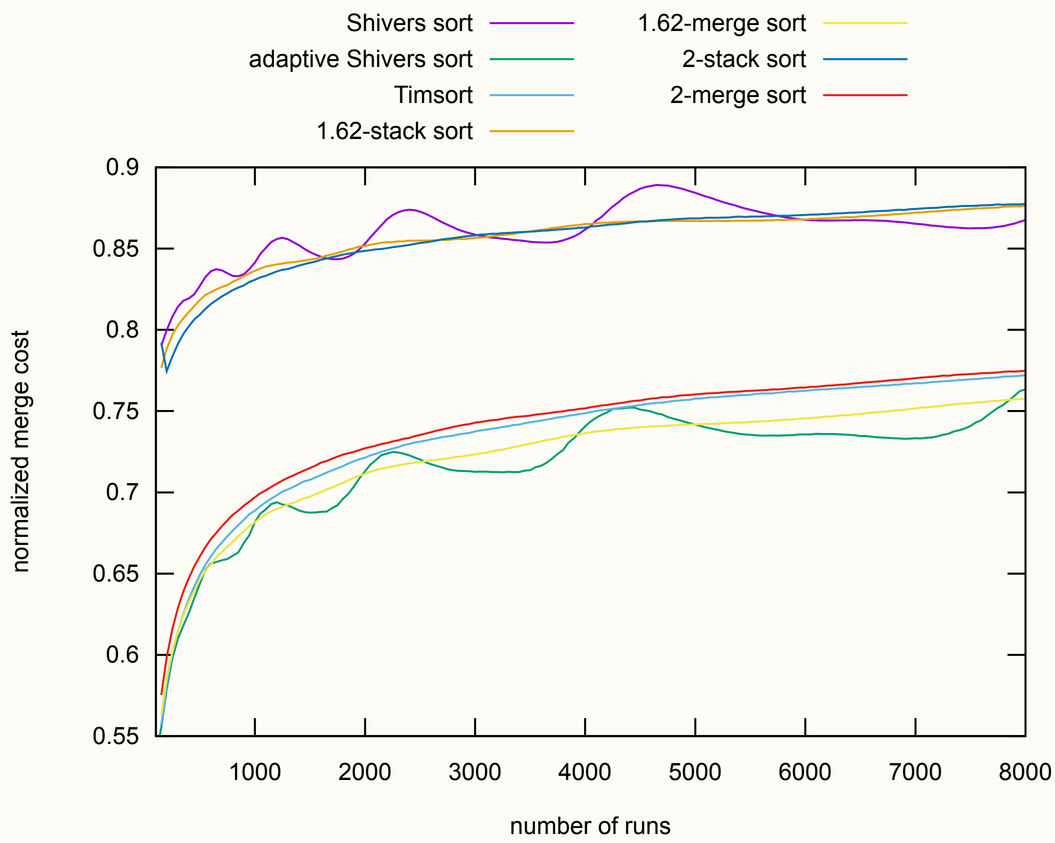
There is 3-aware algorithm achieving upper bounds and worst-case lower bounds equal to $n \log m$.

Moreover, the upper bounds have the form $(1 + o(1))n\mathcal{H}$, where $\mathcal{H}$ is the entropy-based *optimum, non-stable* merge-cost.

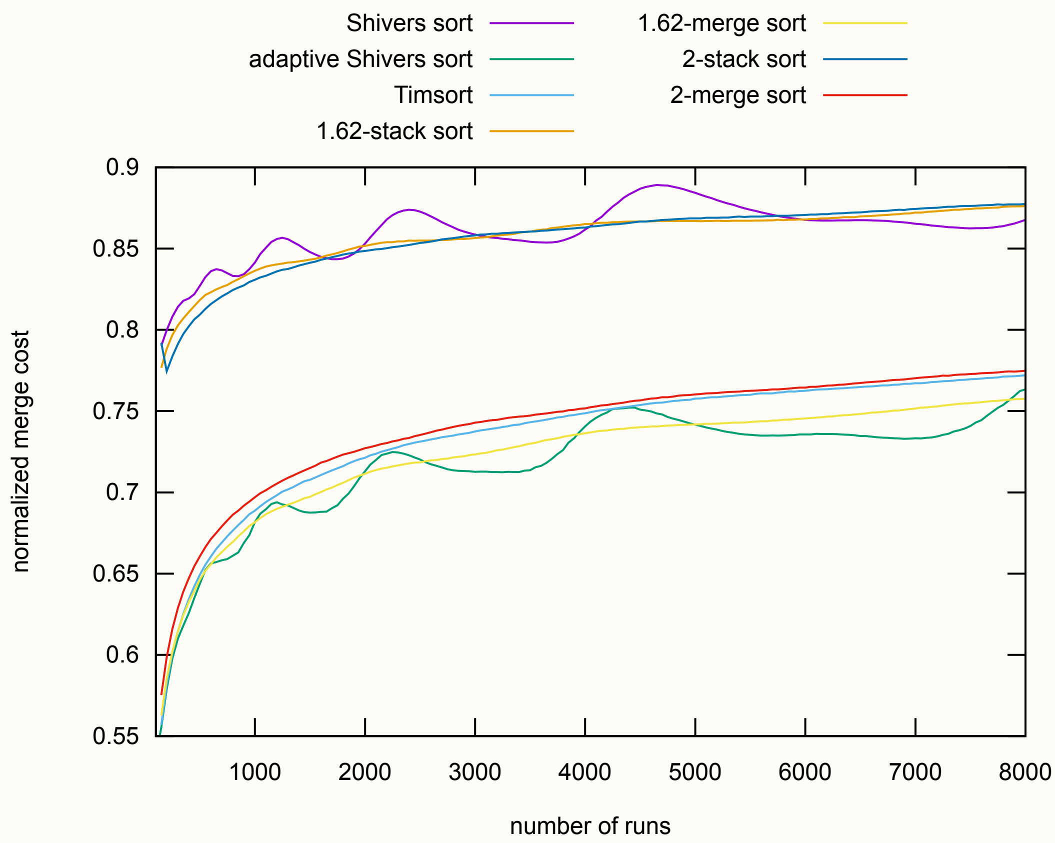
THEOREM (JUGE, P.C.)

The 1.5 lower bound for TimSort is asymptotically tight.

Experimental results

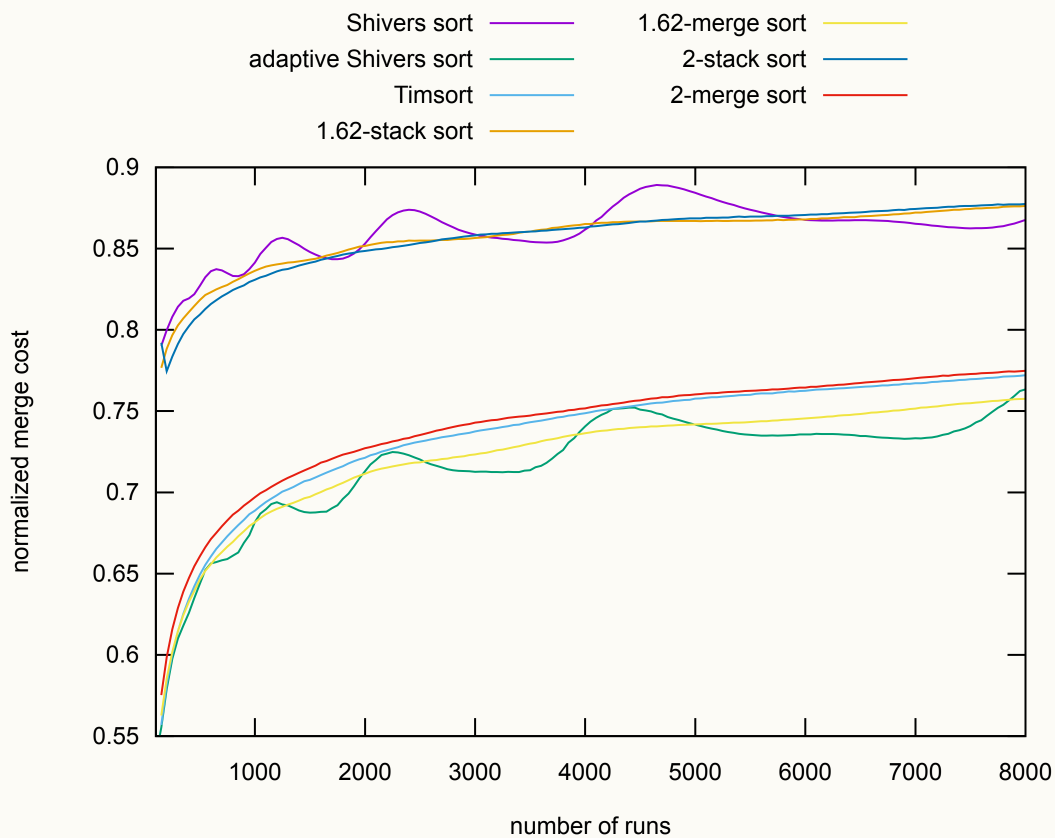


Experimental results



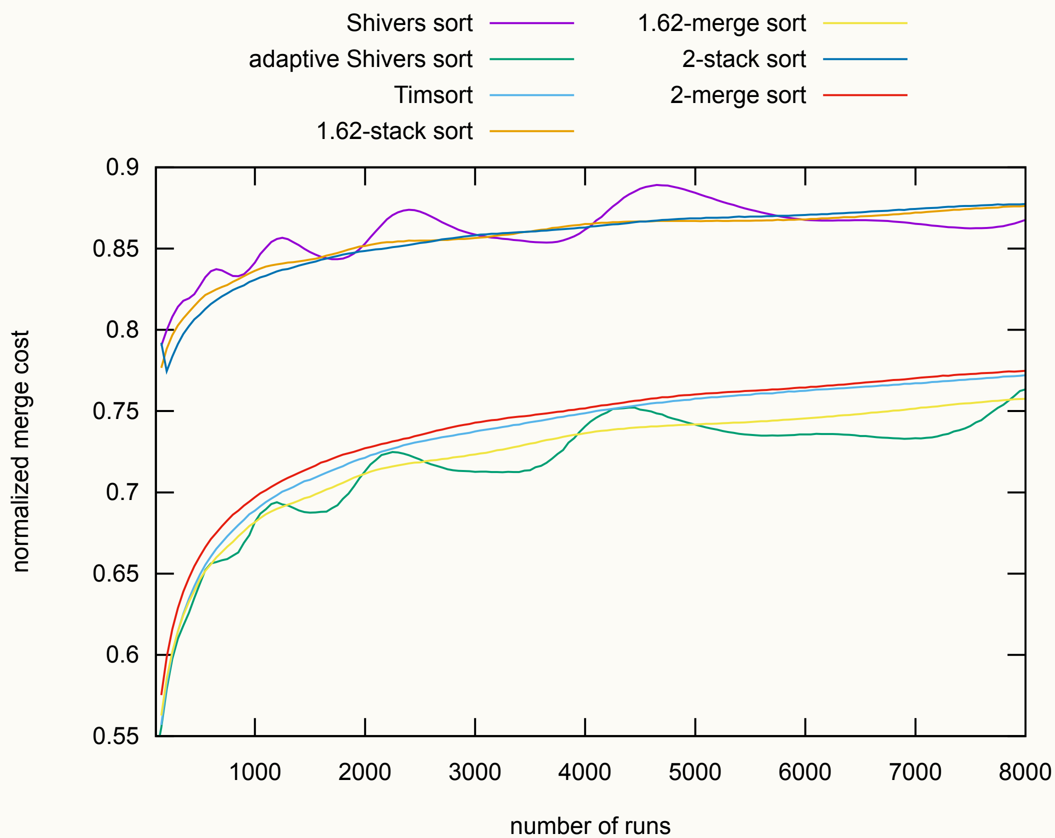
We generate *m* runs:

Experimental results



We generate *m* runs: with probability 0.95 the run has uniformly random length from 1 to 100, and

Experimental results



We generate *m* runs: with probability 0.95 the run has uniformly random length from 1 to 100, and with probability 0.05 the run has uniformly random length from 10^4 to 10^5.

Future Work / Open Questions?

- Would it be worthwhile/possible to collect real-world data to choose the best-in-practice merge sort algorithm? E.g., with only a small overhead, this could be done globally on smartphones.
- Explain the behavior of the algorithms during the simulation.