Complexity of the Adaptive ShiversSort Algorithm and of its sibling TimSort

Vincent Jugé

LIGM – Université Paris-Est Marne-la-Vallée, ESIEE, ENPC & CNRS

07/01/2020
Contents

1 Efficient Merge Sorts

2 TimSort

3 Adaptive ShiversSort
Sorting data

MergeSort has a worst-case time complexity of $O(n \log n)$. Can we do better? No!

Proof: There are $n!$ possible reorderings. Each element comparison gives a 1-bit information, thus $\log_2(n!) \sim n \log_2(n)$ tests are required.
Sorting data

MergeSort has a **worst-case time complexity** of $O(n \log(n))$

Can we do better?
Sorting data

MergeSort has a **worst-case time complexity** of $\mathcal{O}(n \log(n))$

Can we do better? **No!**

**Proof:**
- There are $n!$ possible reorderings
- Each element comparison gives a 1-bit information
- Thus $\log_2(n!) \sim n \log_2(n)$ tests are required
Cannot we ever do better?

In some cases, we should...

\[ \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array} \]
Let us do better!

 Chunk your data in **non-decreasing runs**
Let us do better!

5 runs of lengths 4, 3, 1, 2 and 2

 Chunk your data in non-decreasing runs

New parameters: Number of runs ($\rho$) and their lengths ($r_1, \ldots, r_\rho$)
Let us do better!

5 runs of lengths 4, 3, 1, 2 and 2

Chunk your data in **non-decreasing runs**

New parameters: **Number of runs** ($\rho$) and their **lengths** ($r_1, \ldots, r_\rho$)

**Run-length entropy**: $H = \sum_{i=1}^{\rho} (r_i/n) \log_2(n/r_i) \\ \leq \log_2(\rho) \leq \log_2(n)$
Let us do better!

5 runs of lengths 4, 3, 1, 2 and 2

0 2 2 4 0 1 5 4 1 3 2 3

1. Chunk your data in non-decreasing runs
2. New parameters: Number of runs ($\rho$) and their lengths ($r_1, \ldots, r_\rho$)

Run-length entropy: $\mathcal{H} = \sum_{i=1}^{\rho} \left( \frac{r_i}{n} \right) \log_2 \left( \frac{n}{r_i} \right) \leq \log_2 (\rho) \leq \log_2 (n)$

Theorem [7]
TimSort has a worst-case time complexity of $\mathcal{O}(n + n \mathcal{H})$
Let us do better!

5 runs of lengths 4, 3, 1, 2 and 2

0 2 2 4 0 1 5 4 1 3 2 3

1. Chunk your data in **non-decreasing runs**
2. New parameters: **Number of runs** ($\rho$) and their **lengths** ($r_1, \ldots, r_{\rho}$)

**Run-length entropy:** $\mathcal{H} = \sum_{i=1}^{\rho} (r_i / n) \log_2(n / r_i) \\ \leq \log_2(\rho) \leq \log_2(n)$

---

**Theorem [7]**

**TimSort** has a **worst-case time complexity** of $O(n + n \mathcal{H})$

---

**We cannot do better than $\Omega(n + n \mathcal{H})$!**

- Reading the whole input requires a time $\Omega(n)$
- There are $X$ possible reorderings, with $X \geq 2^{1-\rho} \binom{n}{r_1 \ldots r_{\rho}} \geq 2^n \mathcal{H}/2$

---
Contents

1. Efficient Merge Sorts
2. TimSort
3. Adaptive ShiversSort
A brief history of TimSort

- 2001
- 2002
- 2003
- 2004
- 2005
- 2006
- 2007
- 2008
- 2009
- 2010
- 2011
- 2012
- 2013
- 2014
- 2015
- 2016
- 2017
- 2018
- 2019
A brief history of TimSort

1 Invented by Tim Peters\textsuperscript{[3]}

\textsuperscript{1} Bugs uncovered in Python & Java implementations\textsuperscript{[5,7]}

Refined worst-case analysis\textsuperscript{[7]} – TimSort works in time $O(n + nH)$

$\text{st}$ worst-case complexity analysis\textsuperscript{[6]} – TimSort works in time $O(n \log n)$

Standard algorithm ———————— for non-primitive arrays in Android, Java, Octave

Standard algorithm in Python

V. Jugè

Complexity of the Adaptive ShiversSort Algorithm
A brief history of TimSort

1. Invented by Tim Peters\cite{3}
2. Standard algorithm in Python
A brief history of TimSort

1. Invented by Tim Peters[^3]
2. Standard algorithm in Python
3. ———— for non-primitive arrays in Android, Java, Octave
A brief history of TimSort

1. Invented by Tim Peters\[^3\]
2. Standard algorithm in Python
3. ————————— for non-primitive arrays in Android, Java, Octave
4. 1\(^{st}\) worst-case complexity analysis\[^6\] – TimSort works in time $O(n \log n)$
A brief history of TimSort

1. Invented by **Tim Peters**\[^{[3]}\]
2. Standard algorithm in **Python**
3. ———— for non-primitive arrays in **Android, Java, Octave**
4. 1\(^{st}\) worst-case complexity analysis\[^{[6]}\] – TimSort works in time $O(n \log n)$
5. Refined worst-case analysis\[^{[7]}\] – TimSort works in time $O(n + n H)$

\[^{[3]}\] Complexity of the Adaptive ShiversSort Algorithm

V. Jugé
Invented by Tim Peters\textsuperscript{[3]}

Standard algorithm in Python

---------- for non-primitive arrays in Android, Java, Octave

1\textsuperscript{st} worst-case complexity analysis\textsuperscript{[6]} – TimSort works in time $O(n \log n)$

Refined worst-case analysis\textsuperscript{[7]} – TimSort works in time $O(n + n H)$

Bugs uncovered in Python & Java implementations\textsuperscript{[5,7]}
The principles of TimSort and of adaptive ShiversSort (1/2)

Algorithm based on merging adjacent runs

\[
\begin{array}{cccccc}
0 & 2 & 2 & 4 & 0 & 1 & 5 \\
\hline
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 0 & 1 & 2 & 2 & 4 & 5 \\
\end{array}
\]

Policy for choosing runs to merge:
▶ depends on run lengths only

Complexity analysis:
Evaluate the total merge cost
Forget array values and only work with run lengths

V. Jugé

Complexity of the Adaptive ShiversSort Algorithm
The principles of TimSort and of adaptive ShiversSort (1/2)

Algorithm based on merging adjacent runs  

Stable algorithm

(good for composite types)

0 2 2 4 0 1 5

Run merging algorithm: standard + many optimizations

- Time: $O(k + \ell)$
- Memory: $O(\min(k, \ell))$

Merge cost: $k + \ell$

Policy for choosing runs to merge:

- depends on run lengths only

Complexity analysis:

Evaluate the total merge cost

Forget array values and only work with run lengths
The principles of TimSort and of adaptive ShiversSort (1/2)

Algorithm based on **merging** adjacent runs  

👉 **Stable** algorithm  
(good for **composite** types)

$\equiv$

#### Run merging algorithm: standard + many optimizations

- time $O(k + \ell)$
- memory $O(\min(k, \ell))$

Merge cost: $k + \ell$

---

V. Jugé

Complexity of the Adaptive ShiversSort Algorithm
The principles of TimSort and of adaptive ShiversSort (1/2)

Algorithm based on **merging** adjacent runs

**Stable** algorithm

(good for **composite** types)

Run merging algorithm: standard + many optimizations

- time $\mathcal{O}(k + \ell)$
- memory $\mathcal{O}(\min(k, \ell))$

**Merge cost:** $k + \ell$

**Policy** for choosing runs to merge:

- depends on **run lengths** only

\[ \begin{array}{ccccccc}
0 & 2 & 2 & 4 & 0 & 1 & 5 \\
\hline
k & \ell
\end{array} \equiv \begin{array}{c}
4 \\
3
\end{array} \]

\[ \begin{array}{ccccccc}
0 & 0 & 1 & 2 & 2 & 4 & 5 \\
\hline
\end{array} \equiv \begin{array}{c}
7
\end{array} \]
The principles of TimSort and of adaptive ShiversSort (1/2)

Algorithm based on merging adjacent runs

- **Stable algorithm**
- (good for composite types)

![Merging diagram](image)

1. **Run merging** algorithm: standard + many optimizations
   - time $O(k + \ell)$
   - memory $O(\min(k, \ell))$
   \[
   \text{Merge cost: } k + \ell
   \]

2. **Policy** for choosing runs to merge:
   - depends on run lengths only

3. **Complexity analysis**:
   - Evaluate the total merge cost
   - Forget array values and only work with run lengths
Some results about merge costs

**Best-case** merge costs:

- Every algorithm has a **best-case** merge cost of at least $n\mathcal{H}^{[4,10]}$

**Worst-case** merge costs:

![Diagram showing merge cost range from 0 to $n\mathcal{H}$]
Some results about merge costs

**Best-case** merge costs:
- Every algorithm has a **best-case** merge cost of at least $n \mathcal{H}^{[4, 10]}$

**Worst-case** merge costs:
- TimSort has a **worst-case** merge cost of $3/2 n \mathcal{H} + O(n)^{[7, 9]}$
Some results about merge costs

Best-case merge costs:
- Every algorithm has a best-case merge cost of at least $n \mathcal{H}^{[4,10]}

Worst-case merge costs:
- TimSort has a worst-case merge cost of $3/2 n \mathcal{H} + \mathcal{O}(n)^{[7,9]}
- Adaptive ShiversSort has a worst-case merge cost of $n \mathcal{H} + \mathcal{O}(n)^{[10]}$
Contents

1 Efficient Merge Sorts
2 TimSort
3 Adaptive ShiversSort
The principles of adaptive ShiversSort and of TimSort (2/2)

0 | 2 | 2 | 4 | 0 | 1 | 5 | 4 | 1 | 3 | 2 | 3
≡ 4 | 3 | 1 | 2 | 2

Run merge policy:
- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

\[
\begin{array}{cccccccc}
0 & 2 & 2 & 4 & 0 & 1 & 5 & 4 & 1 & 3 & 2 & 3
\end{array}
\]

\[
\begin{array}{cccc}
4 & 3 & 1 & 2 & 2
\end{array}
\]

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1\textsuperscript{st} and 2\textsuperscript{nd} runs
  3. merge the top 2\textsuperscript{nd} and 3\textsuperscript{rd} runs
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

\[
\begin{array}{cccccccc}
0 & 2 & 2 & 4 & 0 & 1 & 5 & 4 & 1 & 3 & 2 & 3
\end{array}
\equiv
\begin{array}{cccc}
4 & 3 & 1 & 2 & 2
\end{array}
\]

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

0 2 2 4 0 1 5 4 1 3 2 3 ≡ 4 3 1 2 2

V. Jugé

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs

Complexity of the Adaptive ShiversSort Algorithm
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

\[
\begin{align*}
0 & \quad 2 & \quad 2 & \quad 4 & \quad 0 & \quad 1 & \quad 5 & \quad 4 & \quad 1 & \quad 3 & \quad 2 & \quad 3 \\
\end{align*}
\]

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1\textsuperscript{st} and 2\textsuperscript{nd} runs
  3. merge the top 2\textsuperscript{nd} and 3\textsuperscript{rd} runs
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

0 2 2 4 0 1 5 4 1 3 2 3 ≡ 4 3 1 2 2

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

0 2 2 4 0 1 5 4 1 3 2 3

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

0 2 2 4 0 1 5 4 1 3 2 3

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs

STACK
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

```
0 2 2 4 0 1 5 4 1 3 2 3
```

```
0 2 2 4 0 1 4 5 1 3 2 3
```

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1\(^{st}\) and 2\(^{nd}\) runs
  3. merge the top 2\(^{nd}\) and 3\(^{rd}\) runs
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

```
0 2 2 4 0 1 5 4 1 3 2 3
```

```
0 2 2 4 0 1 4 5 1 3 2 3
```

```
0 0 1 2 2 4 4 5 1 3 2 3
```

```
4 3 1 2 2
```

```
4 4 2 2
```

```
8 2 2
```

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1\textsuperscript{st} and 2\textsuperscript{nd} runs
  3. merge the top 2\textsuperscript{nd} and 3\textsuperscript{rd} runs
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

```
<table>
<thead>
<tr>
<th>0 2 2 4</th>
<th>0 1 5 4</th>
<th>1 3 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 2 4</td>
<td>0 1 4 5</td>
<td>1 3 2 3</td>
</tr>
<tr>
<td>0 0 1 2 2 4 4 5 1 3 2 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>4 3 1 2 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 4 2 2</td>
</tr>
<tr>
<td>8 2 2</td>
</tr>
</tbody>
</table>
```

STACK

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1\(^{st}\) and 2\(^{nd}\) runs
  3. merge the top 2\(^{nd}\) and 3\(^{rd}\) runs

V. Jugé

Complexity of the Adaptive ShiversSort Algorithm
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

0 2 2 4 0 1 5 4 1 3 2 3

0 2 2 4 0 1 4 5 1 3 2 3

0 0 1 2 2 4 4 5 1 3 2 3

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1st and 2nd runs
  3. merge the top 2nd and 3rd runs
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

Run merge policy:
- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1\textsuperscript{st} and 2\textsuperscript{nd} runs
  3. merge the top 2\textsuperscript{nd} and 3\textsuperscript{rd} runs
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

```
0 2 2 4 0 1 5 4 1 3 2 3
```

```
4 3 1 2 2
```

```
0 2 2 4 0 1 4 5 1 3 2 3
```

```
4 4 2 2
```

```
0 0 1 2 2 4 4 5 1 3 2 3
```

```
8 2 2
```

```
0 0 1 2 2 4 4 5 1 2 3 3
```

```
8 4
```

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1\textsuperscript{st} and 2\textsuperscript{nd} runs
  3. merge the top 2\textsuperscript{nd} and 3\textsuperscript{rd} runs

V. Jugé

Complexity of the Adaptive ShiversSort Algorithm
The principles of adaptive ShiversSort and of TimSort (2/2)

Discovered runs:

Run merge policy:

- Maintain a stack of runs
- Until the array is sorted, either:
  1. discover & push a new run onto the stack
  2. merge the top 1\textsuperscript{st} and 2\textsuperscript{nd} runs
  3. merge the top 2\textsuperscript{nd} and 3\textsuperscript{rd} runs
Intermezzo: Intelligent design & amortized analysis

Key ideas:
- Each run \( r \) pays its share of the total merge cost
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run $r$ pays
  - $O(r)$ to enter the stack (entry phase)
  - $r$ to increase its bit length (growth phase)

Bit length of $r$: $\ell = \lfloor \log_2(r) \rfloor$

Cost analysis:

- Each run $r$ pays
  - $O(r)$ during its own run entry phase
  - at most $r\lceil \log_2(n/r) \rceil$ during the growth phases

- Total merge cost of $nH + O(n)$
Intermezzo: Intelligent design & amortized analysis

Key ideas:
- Each run $r$ pays
  - $\mathcal{O}(r)$ to enter the stack (entry phase)
  - $r$ to increase its bit length (growth phase)
    
    bit length of $r$: $\ell = \lfloor \log_2(r) \rfloor$

- Entry phase:

Cost analysis:
- Each run $r$ pays
  - $\mathcal{O}(r)$ during its own run entry phase
  - at most $r \lfloor \log_2(n/r) \rfloor$ during the growth phases

- Total merge cost of $nH + \mathcal{O}(n)$
Intermezzo: Intelligent design & amortized analysis

Key ideas:
- Each run $r$ pays
  - $O(r)$ to enter the stack (entry phase)
  - $r$ to increase its bit length (growth phase)
- Entry phase:
  - bit length of $r$: $\ell = \lfloor \log_2(r) \rfloor$

Cost analysis:
- Each run $r$ pays
  - $O(r)$ during its own run entry phase
  - at most $r \lceil \log_2(n/r) \rceil$ during the growth phases
- Total merge cost of $n \mathcal{H} + O(n)$
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run $r$ pays
  - $O(r)$ to enter the stack (entry phase)
  - $r$ to increase its bit length (growth phase)

- Entry phase:

Cost analysis:

- Each run $r$ pays
  - $O(r)$ during its own run entry phase
  - at most $r \lfloor \log_2(n/r) \rfloor$ during the growth phases

- Total merge cost of $nH + O(n)$
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run $r$ pays
  - $O(r)$ to enter the stack (entry phase)
  - $r$ to increase its bit length (growth phase)
  
    \[
    \text{bit length of } r: \ell = \lfloor \log_2(r) \rfloor
    \]

- Entry phase: ensure that
  - $r$ pays for every merge
  - $(r_i)_{i \geq 1}$ has exponential decay when $r$ is pushed
  - runs smaller than $r$ are merged

Cost analysis:

- Each run $r$ pays
  - $O(r)$ during its own run entry phase
  - at most $r \lceil \log_2(n/r) \rceil$ during the growth phases

- Total merge cost of $nH + O(n)$
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run \( r \) pays
  - \( \mathcal{O}(r) \) to enter the stack (entry phase)
  - \( r \) to increase its bit length (growth phase)
    
    bit length of \( r \): \( \ell = \lfloor \log_2(r) \rfloor \)

- Entry phase: ensure that
  - \( r \) pays for every merge
  - \( (r_i)_{i \geq 1} \) has exponential decay when \( r \) is pushed
  - runs smaller than \( r \) are merged

- Growth phase: ensure that
  - \( r_i \) and \( r_{i+1} \) are merged only if their bit lengths are equal

Cost analysis:

- Each run \( r \) pays
  - \( \mathcal{O}(r) \) during its own run entry phase
  - at most \( r \lceil \log_2(n/r) \rceil \) during the growth phases

- Total merge cost of \( n\mathcal{H} + \mathcal{O}(n) \)
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run \( r \) pays
  - \( \mathcal{O}(r) \) to enter the stack (entry phase)
  - \( r \) to increase its bit length (growth phase)
    - bit length of \( r \): \( \ell = \lfloor \log_2(r) \rfloor \)

- **Entry phase**: ensure that
  - \( r \) pays for every merge
  - \( (\ell_i)_{i \geq 1} \) is decreasing when \( r \) is pushed
  - runs \( r_i \) with \( \ell_i \leq \ell \) are merged

- **Growth phase**: ensure that
  - \( r_i \) and \( r_{i+1} \) are merged only if \( \ell_i = \ell_{i+1} \)

Cost analysis:

- Each run \( r \) pays
  - \( \mathcal{O}(r) \) during its own run entry phase
  - at most \( r \lfloor \log_2(n/r) \rfloor \) during the growth phases

- **Total merge cost** of \( n \mathcal{H} + \mathcal{O}(n) \)
Intermezzo: Intelligent design & amortized analysis

Key ideas:

- Each run $r$ pays
  - $\mathcal{O}(r)$ to enter the stack (entry phase)
  - $r$ to increase its bit length (growth phase)

- Bit length of $r$: $\ell = \lfloor \log_2(r) \rfloor$

- Entry phase: ensure that
  - $r$ pays for every merge
  - $(\ell_i)_{i \geq 1}$ is decreasing when $r$ is pushed
  - runs $r_i$ with $\ell_i \leq \ell$ are merged

- Growth phase: ensure that
  - $r_i$ and $r_{i+1}$ are merged only if $\ell_i = \ell_{i+1}$

Cost analysis:

- Each run $r$ pays
  - $\mathcal{O}(r)$ during its own run entry phase
  - at most $r \lceil \log_2(n/r) \rceil$ during the growth phases

- Total merge cost of $nH + \mathcal{O}(n)$
The details of Adaptive ShiversSort

Choice rules for options

1. discover & push a new run length onto the stack
2. merge the top 1st and 2nd runs
3. merge the top 2nd and 3rd runs

Choice algorithm

\[
\begin{align*}
\text{if } \ell_h & \geq \ell_{h-2} \text{ or } \ell_{h-1} \geq \ell_{h-2}: \text{ choose } 3 \\
\text{else if } \ell_h & \geq \ell_{h-1}: \text{ choose } 2 \\
\text{else: choose } 1 \text{ (or } 2 \text{ if } 1 \text{ is unavailable)}
\end{align*}
\]

where \( \ell_i = \lceil \log_2(r_i) \rceil \)
The details of Adaptive ShiversSort

**Choice rules** for options

1. discover & push a new run length onto the stack
2. merge the top 1\textsuperscript{st} and 2\textsuperscript{nd} runs
3. merge the top 2\textsuperscript{nd} and 3\textsuperscript{rd} runs

**Choice algorithm**

\[
\begin{align*}
\text{if } & \ell_h \geq \ell_{h-2} \text{ or } \ell_{h-1} \geq \ell_{h-2}: \text{ choose } (3) \\
\text{else if } & \ell_h \geq \ell_{h-1}: \text{ choose } (2) \\
\text{else: } & \text{ choose (1) (or (2) if (1) is unavailable)}
\end{align*}
\]

where \( \ell_i = \lfloor \log_2(r_i) \rfloor \)
The details of Adaptive ShiversSort

Choice rules for options

1. discover & push a new run length onto the stack
2. merge the top 1\textsuperscript{st} and 2\textsuperscript{nd} runs
3. merge the top 2\textsuperscript{nd} and 3\textsuperscript{rd} runs

Choice algorithm

\[
\begin{align*}
\text{if } \ell_h &\geq \ell_{h-2} \text{ or } \ell_{h-1} \geq \ell_{h-2}: \text{ choose 3} \\
\text{else if } \ell_h &\geq \ell_{h-1}: \text{ choose 2} \\
\text{else: } \text{ choose 1 (or 2 if 1 is unavailable)}
\end{align*}
\]

where \( \ell_i = \lfloor \log_2(r_i) \rfloor \)

Bit-length constraints:

- \( \ell_1 > \ell_2 > \ldots > \ell_{h-2} \geq \ell_{h-1} \) (induction)
- \( \ell_1 > \ell_2 > \ldots > \ell_h \) on run push
- \( \ell_{h-1} \geq \ell_h \) and \( \ell_{h-2} > \ell_h \) during growth (induction)
The details of Adaptive ShiversSort

Choice rules for options
1. discover & push a new run length onto the stack
2. merge the top 1\textsuperscript{st} and 2\textsuperscript{nd} runs
3. merge the top 2\textsuperscript{nd} and 3\textsuperscript{rd} runs

Choice algorithm

\begin{align*}
\text{if } & \ell_h \geq \ell_{h-2} \text{ or } \ell_{h-1} \geq \ell_{h-2}: \text{ choose } 3 \\
\text{else if } & \ell_h \geq \ell_{h-1}: \text{ choose } 2 \\
\text{else: } & \text{ choose } 1 \text{ (or } 2 \text{ if } 1 \text{ is unavailable)}
\end{align*}

where \( \ell_i = \lceil \log_2(r_i) \rceil \)

Bit-length constraints:

- \( \ell_1 > \ell_2 > \ldots > \ell_{h-2} \geq \ell_{h-1} \) (induction)
- \( \ell_1 > \ell_2 > \ldots > \ell_h \) on run push
- \( \ell_{h-1} \geq \ell_h \) and \( \ell_{h-2} > \ell_h \) during growth (induction)

END OF PROOF!
Conclusion

- **TimSort** is good in practice and in theory: $O(n + n\mathcal{H})$ merge cost
- **Adaptive ShiversSort** is better than and very similar to TimSort
Conclusion

- **TimSort** is good in practice and in theory: $O(n + \log n)$ merge cost
- **Adaptive ShiversSort** is better than and very similar to TimSort

Some references:


thank you