Department of Computer Science and Engineering
The Chinese University of Hong Kong

CSCI2100S DATA STRUCTURES

Spring 2011

Seminar 1: Timsort

Last updated: **18/02/2011**

Tang Wai Chung, Matthew

Contents

Facts about Timsort

- **Hybrid** sorting algorithm – merge sort x insertion sort
- designed for **real-world data**
  - contrast to random data in quicksort
- Inverted by **Tim** Peters in 2002
  - use in Python language
  - adopt by Java SE 7 & Android
- Important features
  - adaptive, stable, natural merge sort
  - insertion sort for small cut-off

Source: Wikipedia, timsort.txt

Advantages

- **Adaptive** – takes advantage of existing order in its input.
  - Super fast on partially ordered arrays
  - Reasonable runtime on random data – reported that twice as slow as quicksort.
- **Stable**
  "Back on Earth, among Python users the most frequent complaint I've heard is that list.sort() isn't stable."
- No bad cases: $O(N \log N)$
  Only good cases: $N - 1$ compares.

Terminologies

- **Run** – ordered sequence of items, either
  - (monotonic) ascending, or
    $A[0] <= A[1] <= A[2] <= \ldots$
  - (strictly) descending
    $A[0] > A[1] > A[2] > \ldots$
- A run is always at least 2 items long (except the last element alone)
- Merge **Stack**: the last-in, first-out (LIFO) array for arranging the merging sequence of different runs.
- **Galloping**: the exponential search technique that finds a key k in in array A with $|A| = N$ elements with about $2*\lg(N)$ comparisons.

Timsort: Overview

Run Extraction/Boosting

- Compute minrun based on the input array
- Identify runs
- Boost runs to length minrun using binary insertion sort
- Push runs to the merge stack

Merge Collapse

- Merge runs on the top of the stack (2 at a time), while maintain a pre-defined invariant

Merging

- Normal mode: merge as usual, and count the no. of times in a row the winning merge element comes, switch to gallop mode if it is large enough (**MIN_GALLOP**)
- Gallop mode: fast merging by moving a chuck of elements from the array that keeps winning over the normal mode

Computing minrun

- If $N < 64$, minrun = N
  - Use binary insertion sort completely.
  - **Tim: "*It is hard to beat that given the overheads of trying sth fancier*."**
- When N is power of 2 (2^K), minrun is chosen to be 32.
- **Goal**: To make the merges end up perfectly balanced
- Sometimes 32 is not a good choice!
  e.g. 2112 / 32 = 66, 2112 % 32 = 0
  Result: runs of lengths 2048 and 64 to merge

Computing minrun (Cont')

- In the last example, 33 is a better choice.
- In general, the idea is to **avoid**

$$q = N \,/\, \underline{\text{minrun}},\ r = N \,\%\, \underline{\text{minrun}}$$

  - (i) $q = 2^K$ and $r < \underline{\text{minrun}}$
  - (ii) $q > 2^K$ and $r = 0$
- Smart minrun computation:
  - in range (32, 65) s.t. **N/<u>minrun</u> is close to 2^K**
  - **Take the first (most significant) 6 bits of N, and add 1 if any remaining bits are set.**

Computing minrun: Example

$2112_{10} = \underline{100001}000000$

No remaining bits are set.

minrun = 100001 = 33_{10}

$4096_{10} = \underline{1000000}000000$

No remaining bits are set.

minrun = 100000 = 32_{10}

$1000_{10} = \underline{111110}1000$

Some remaining bits are set.

minrun = 111110 + 1 = 63_{10}

<table>
<tr><td>63</td><td>63</td><td>63</td><td>63</td><td>...</td><td>63</td><td>55</td></tr>
<tr><td colspan="2">126</td><td colspan="2">126</td><td></td><td colspan="2">118</td></tr>
<tr><td colspan="4">252</td><td colspan="3">224</td></tr>
<tr><td colspan="7">...</td></tr>
<tr><td colspan="7">1000</td></tr>
</table>

Run Identification

- Given an array, it is easy to find the longest run from its beginning.
  - If the whole array is a run, $N - 1$ comparisons are required.
- When the run is descending, it can be **reversed** to become ascending in $O(N)$ time. HOW?
  - We can assure the stability is not broken during the reversal. WHY?
- If the run found is shorter than minrun, it will be **boosted** to a longer run by performing a binary insertion sort.

Binary Insertion Sort (Recap)

- Process items one at a time
- Use binary search to locate the point of insertion among already and sorted.
  - Try to minimize the no. of comparisons (which is costly in Python)
- Then make **space** for moving larger items one position to the right.
  - This is the **bottleneck**: complexity = $O(N^2)$
- Setting minrun > 64 will incur too many movements in binary insertion sort.

Merge Pattern

- Natural runs are wildly unbalanced.
  e.g. **1 1 2 3 4 1 3 2 4 5 6 7 9 8**
- But we want to keep the merge balanced to minimize data movement.
- Stability constraint: **merge only adjacent runs**
  - 3 consecutive runs: *A*: 10000 *B*: 20000 *C*: 10000
  - If *A*, *B*, *C* contains the same key, we cannot merge *A* with *C* → (*A* + *B*) + *C* or *A* + (*B* + *C*)
- Merging is done on two consecutive runs at a time, in-place ***with some temp memory***.

Merge Stack

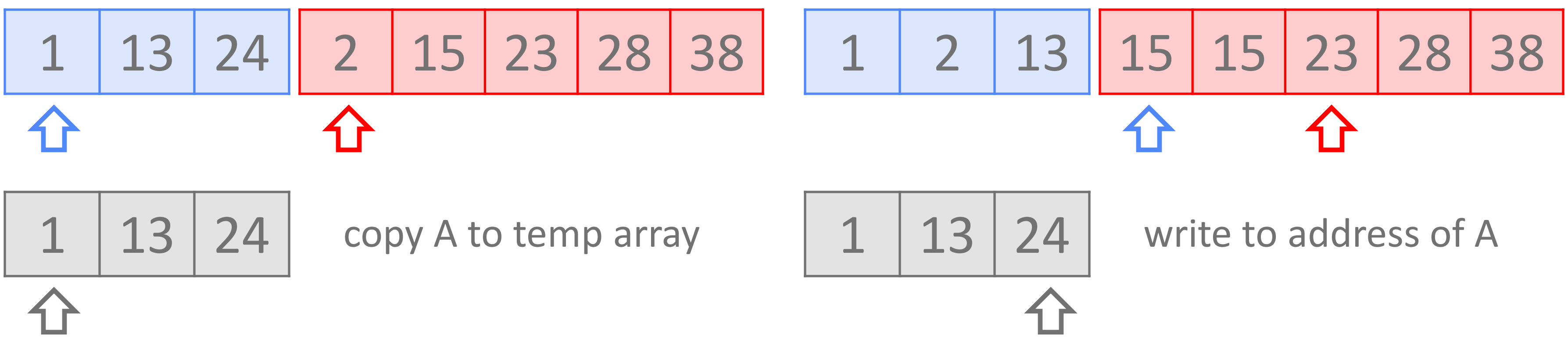
- After a run is identified (or boosted), it will be **pushed** to the merge stack.
- Then the procedure checks whether it should be merged with preceding run(s).
  - Heuristic 1: delay merging as long as possible
  - Heuristic 2: merge recent runs (likely to be remained in cache)
- Just cannot delay forever – memory explosion
- Good comprise to keep 2 invariants on the stack
  - A, B, C are runs on the top of stack
    $|A| > |B| + |C|$ & $|B| > |C|$

Merge Stack

- If $|A| <= |B| + |C|$, smaller of A and C will be merged with B
  - Ties favor C (freshness-in-cache)
  - The new, merged run replaces (A, B) or (B, C).
- Example:
  - A: 30 B: 20 C: 10 → A:30 BC: 30
  - A: 500 B: 400 C: 1000 → AB: 900 C: 1000 violates invariant #2, so repeat.
- **Closing**: when all runs are identified, merge **all** runs from the top to the bottom, 2 at a time.

Pseudo In-place Merging

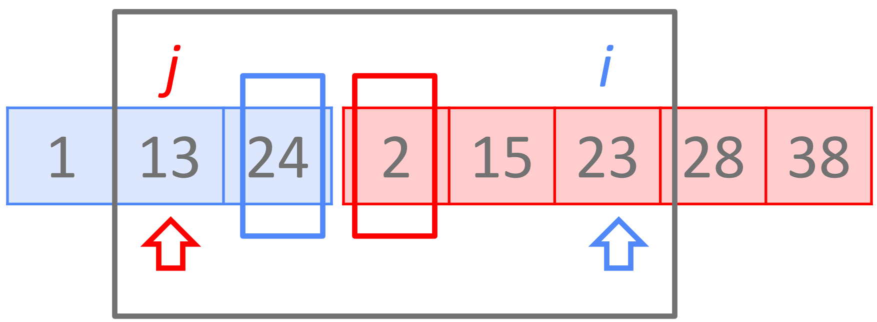
- It is easy to merge two runs *A* and *B* in-place with a temp memory equal of size min(|*A*|, |*B*|).
- If |*A*| < |*B*|, copy *A* to temp array.
- Then we can start merge from left to right, from temp array and *B*.
  - There is **always a free space** in the original area.



Remember to copy unmerged elements from either temp array or B.

Pseudo In-place Merging (Cont')

- There is also a refinement suggested by Tim.
  - See where $B[0]$ should end up in A, say j
    then $A[0]$, $A[1]$, ..., $A[j - 1]$ are in place
  - See where $A[|A| - 1]$ should end up B, say i
    then $B[i + 1]$, $B[i + 2]$, ..., $B[|B|]$ can be ignored.
- Binary search can be used.
  In timsort, ***galloping*** is used instead.



actual merging needed

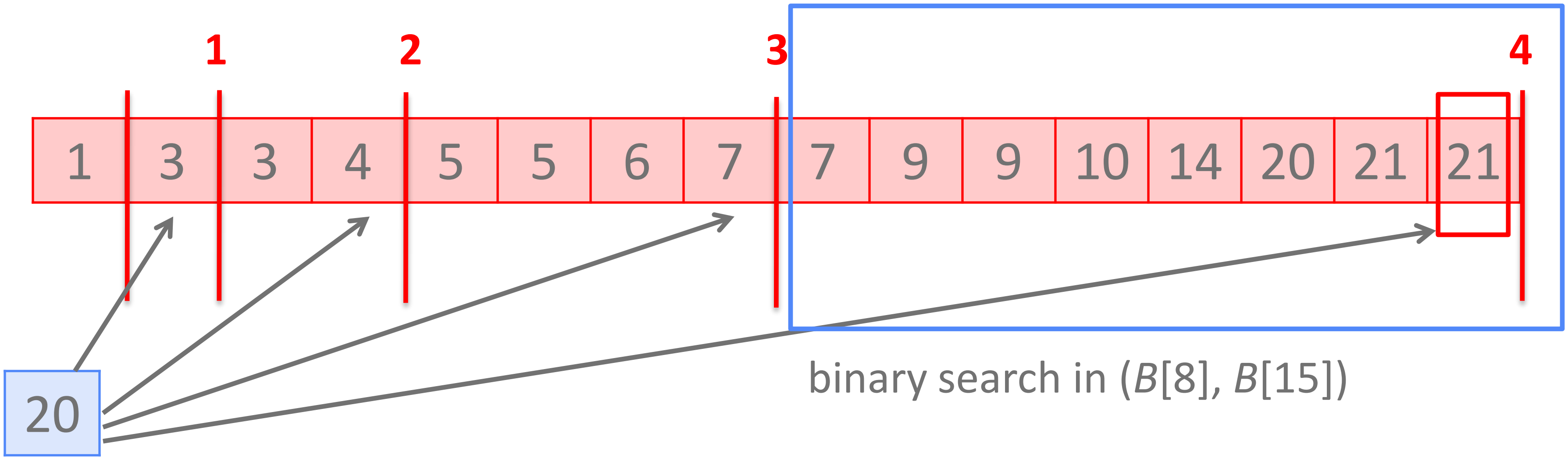
Galloping

- Assume A is the smaller run, when we want to find $A[0]$ in B ...
  - we compare $A[0]$ with $B[0]$, $B[1]$, $B[3]$, $B[7]$, ..., $B[2^j - 1]$
  - until we find a value k s.t. $B[2^{k-1} - 1] < A[0] <= B[2^k - 1]$ with $\lg(|B|)$ comparisons.
- **Why galloping?**
  - Straight binary search takes $\text{ceil}(\lg(|B| + 1))$ comparisons no matter where $A[0]$ belongs.
  - Galloping **stops earlier** when the run is long.

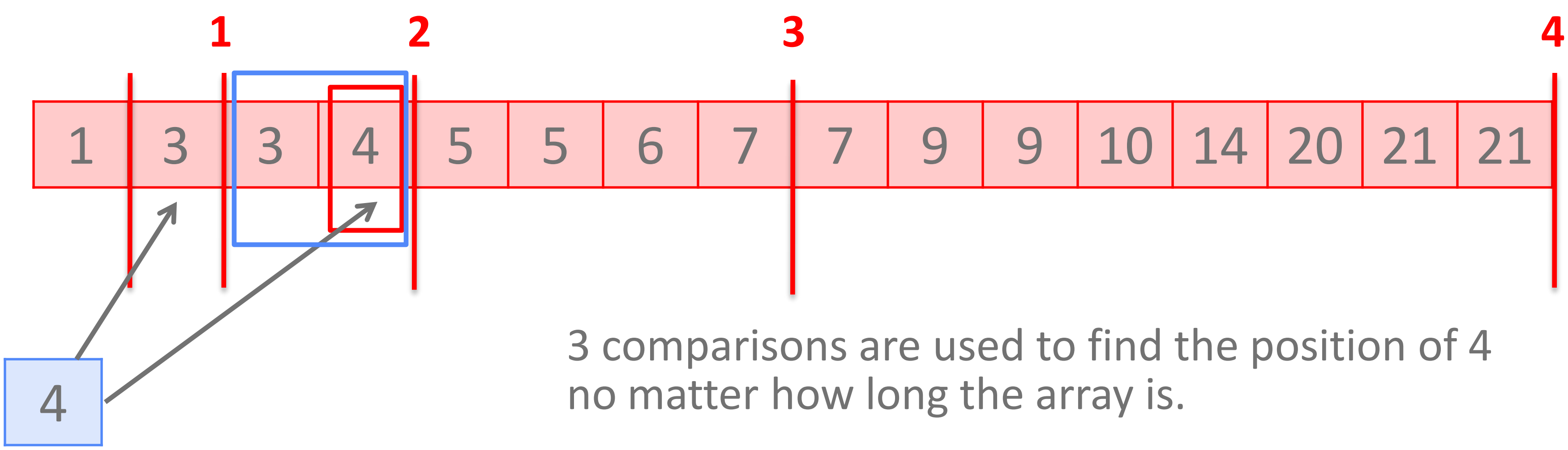
Galloping (Cont')

- When k is found, we can then follow by **a binary search** between $B[2^{k-1} - 1]$ and $B[2^{k} - 1]$.
- When linear search takes (i + 1) comparisons, galloping uses a total of 2 * floor(lg(i)) + 2 comparisons.
  - Galloping wins when after i = 6.
  - It is reasonable to use galloping (during merging) when consecutive wins from one of the arrays is more than **MIN_GALLOP** = 8

Galloping: Example



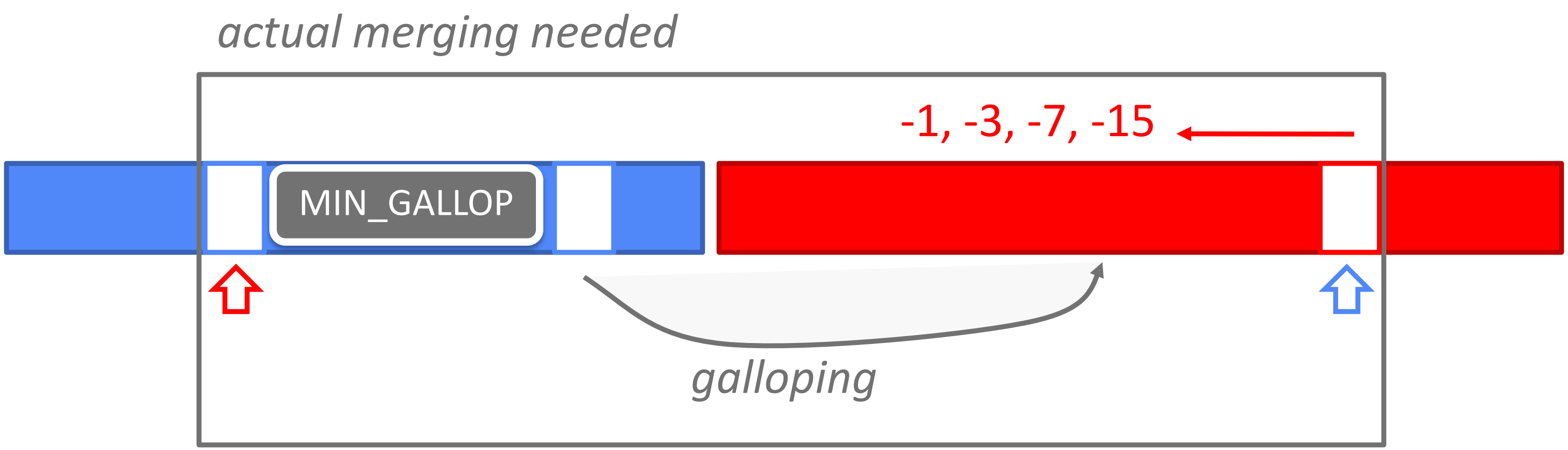
binary search in (B[8], B[15])



3 comparisons are used to find the position of 4
no matter how long the array is.

Galloping Complications

When you want to gallop on *B*, you need to reverse the direction of the search.



- ☒ Starting at the beginning of *B* needs many comparisons.
- ☑ Starting at the end of *B*, try jumping with index -1, -3, -7, -15.

Timsort: Algorithm

```
/* input: array (base address) A with N items */
minrun = compute_minrun(N);
nrem = N;
S = stack_create();
do {
        r = count_run(A);
        if (run is descending) reverse_run(A, r);
        if (r < minrun) boost_run(A);
        stack_push(S, A, max(r, minrun));
        merge_collapse(S);
        A += r;
        nrem -= r;
} while (nrem);
merge_force_collapse(S);
```

```
if (invariant #1 & #2 fail)
   merge()
```

```
na = na – gallop_right()
nb = gallop_left()
if (na <= nb)
   merge_lo();
else
   merge_hi();
```

Seminar Prog. Asgn. 1

- Simplified Timsort
  - Bottom-up merge sort on runs with length <u>minrun</u>
    - No merging stack required
    - Boosting by binary insertion sort
  - Left-to-right in-place merging with galloping
    - Allocate temp array with size $|A|$.
    - Switch to galloping mode when needed.
- Comparison with standard merge sort implementation (provided)

Seminar Prog. Asgn. 1 : Assessment

- Implementation/Correctness (75%)
  - Run boosting and merging loops (25%)
  - Pseudo In-place merging (25%)
  - Galloping (left & right) (25%)
- Performance (25%)
  - speed-up over standard merge sort, competition