Notes: Bucket Sort, Radix Sort, Borůvka's Step, and Fredman–Tarjan Step

Prepared for: Algorithm Study

August 24, 2025

Contents

1 Bucket Sort

1.1 Idea

Bucket sort (a.k.a. bin sort) distributes the input elements into several buckets. Each bucket is then sorted (often with insertion sort or another stable sort). Finally, the buckets are concatenated in order to produce the sorted output. Bucket sort is particularly effective when input is drawn from a uniform distribution over a known range.

1.2 Pseudocode

Algorithm 1 BucketSort(A, k) — sort array $A[1..n]$ into k buckets

Require: array $A[1..n]$, number of buckets k, and a bucket-mapping function bucketIndex(x)
Ensure: A sorted in non-decreasing order

```
Create array of buckets B[0..k − 1], each initially empty
for i ← 1 to n do
    b ← bucketIndex(A[i])                ▷ map key to a bucket index
    insert A[i] into list B[b]
end for
for j ← 0 to k − 1 do
    sort list B[j] (e.g. insertion sort)
end for
Concatenate lists B[0], B[1], . . . , B[k − 1] to form the output
```

1.3 Complexity

- **Average-case time:** $O(n+k)$ when the input is approximately uniformly distributed among the k buckets (choose $k = \Theta(n)$ for linear average time).
- **Worst-case time:** $\Theta(n^2)$ if all elements fall in the same bucket and that bucket is sorted with insertion sort (or $O(n \log n)$ if a comparison sort is used inside each bucket). :contentReference[oaicite:0]index=0
- **Space:** $O(n + k)$.
- **Stability:** Stable if the sort used inside buckets is stable and insertion uses stable insertion.

1.4 When to use

Best when keys are numeric, lie in a known range, and are reasonably uniformly distributed. Common practical uses include floating-point numbers in $[0, 1)$, histogram bucketing, and preliminary pass in some external sorts.

1.5 Illustrative example and diagram

Suppose $A = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]$ and buckets correspond to intervals of length 0.1. After distributing, sort each bucket and concatenate.

1.6 Interesting problem (application)

Problem: Given n real numbers uniformly sampled from $[0, 1)$, design a linear-time algorithm to compute approximate quantiles (e.g., median or 0.1, 0.9 quantiles) to within additive error $1/k$.

Solution sketch:

1. Use k buckets representing intervals $[0, 1/k), [1/k, 2/k), \ldots$.
2. Put each sample into its bucket in $O(n)$ time.
3. Compute prefix counts of buckets to locate which bucket contains the desired rank (e.g., median).
4. Within that bucket, pick the appropriate element (or approximate by uniform interpolation), costing only time proportional to the bucket size.

This yields $O(n + k)$ time; with $k = \sqrt{n}$ or $k = \Theta(n)$ we get a very fast approximate quantile algorithm. (This is a practical technique for streaming/large-data approximate quantiles.)

2 Radix Sort

2.1 Idea

Radix sort orders keys by processing individual digits (or groups of bits) from least significant digit (LSD) to most significant digit (MSD) or vice versa. Each pass is a stable distribution sort (commonly counting sort) on one digit position. Radix sort is especially effective for fixed-width integer keys or fixed-length strings.

2.2 Two variants

- **LSD (Least Significant Digit) radix:** process digits from least significant to most significant; works well for fixed-length integers/strings and requires stable digit-level sort.
- **MSD (Most Significant Digit) radix:** process most significant digit first and recursively sort buckets; suited for variable-length keys (strings).

2.3 Pseudocode (LSD, base b)

Algorithm 2 LSD-Radix-Sort(A, d, b)

Require: array $A[1..n]$ of keys each with d digits in base b, a stable counting sort by digit
Ensure: A sorted

```
for i ← 1 to d do                          ▷ for each digit position from LSD to MSD
    stable-counting-sort-by-digit(A, digit=i, base=b)
end for
```

2.4 Complexity

- **Time:** $O(d \cdot (n + b))$ where d is number of digits and b is base (radix). For fixed-size machine words (e.g., 32-bit integers) and an appropriate choice of base, radix sort runs in $O(n)$. :contentReference[oaicite:1]index=1
- **Space:** $O(n + b)$ for counting arrays / buckets.
- **Stability:** Requires the digit-level sort to be stable (counting sort is commonly used).

2.5 When to use

Sorting large arrays of integers or fixed-length strings where comparison-based $O(n \log n)$ sorts are expensive. Radix is widely used in practice when keys have bounded size (e.g., 32/64-bit integers, phone numbers, IP addresses).

2.6 Interesting problem (application)

Problem: Sort 10^7 32-bit unsigned integers in practice in near-linear time on a single machine.
Solution sketch:

- Use LSD radix with base $b = 2^{16}$ (two passes on 16-bit chunks). Each pass requires counting sort with $b = 65536$ which is feasible in memory and very fast due to locality.

- Complexity: $O(2 \cdot (n+b)) = O(n+b)$. For $n = 10^7$ and $b = 65536$ the dominant cost is linear in n and this is very fast in practice (this is a common trick used in high-performance integer sorts). Real implementations will tune the number of bits per pass for cache efficiency and memory constraints.

3 Borůvka's Step (Borůvka's Algorithm')

3.1 Overview

Borůvka's algorithm (originally from 1926) constructs an MST by repeatedly executing a phase (often called a *Borůvka step*): for every connected component in the current forest, find the minimum-weight outgoing edge, add all those chosen edges to the forest, and contract the newly-formed connected components. Each phase reduces the number of components by at least a constant factor (at least halves it in expectation / often exactly halves in the sense that each component picks an outgoing edge and merges), so only $O(\log n)$ phases are needed.

3.2 Borůvka step pseudocode

Algorithm 3 BoruvkaStep(G = (V,E), forest F)

Require: Graph $G = (V, E)$, a forest F (initially F has all vertices as separate components)
Ensure: forest F' after one Borůvka step

```
Initialize array bestEdge[component] ← NULL
for each edge (u, v, w) ∈ E do
    c_u ← findComponent(u); c_v ← findComponent(v)
    if c_u ≠ c_v then
        if bestEdge[c_u] is NULL or w < weight(bestEdge[c_u]) then bestEdge[c_u] ← (u, v, w)
            if bestEdge[c_v] is NULL or w < weight(bestEdge[c_v]) then bestEdge[c_v] ← (u, v, w)
            end if

            for each component c do
                if bestEdge[c] ≠ NULL then
                    add bestEdge[c] to forest F
                    union the endpoints' components
                end if
            end for
            return updated forest F
```

3.3 Complexity notes

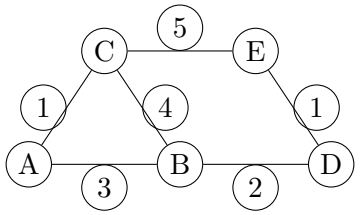
- A single Borůvka step can be implemented by scanning all edges and doing a small number of union/find operations; scanning edges costs $O(m)$ and union-find contributes almost-linear overhead (inverse-Ackermann costs). Therefore one step is $O(m\alpha(n))$ in practice.
- Because the number of components drops quickly (at least by factor 2 in many analyses), there are at most $O(\log n)$ steps, so a straightforward analysis gives $O(m \log n)$ (or $O(m\alpha(n) \log n)$) total time for running Borůvka repeatedly until one component remains; with careful implementation and combination with other techniques one can do better. Borůvka's approach is also extremely parallelizable and works well in distributed settings. :contentReference[oaicite:2]index=2

3.4 When to use

Borůvka's steps are often used as the first phases in faster MST algorithms (they rapidly reduce graph size) and are attractive in parallel/distributed MST implementations because each component

can independently choose its cheapest outgoing edge.

3.5 Illustrative example (small graph)



In one Borůvka step each vertex (component) picks its cheapest incident edge: A picks (A,C,1), B picks (B,D,2), C would pick (A,C,1) or (B,C,4) but sees A picked same, etc. After adding those picked edges and contracting, components shrink.

3.6 Interesting problem (application)

Problem: Use Borůvka steps to speed up MST computation in a very large sparse graph stored on disk (external memory). Describe how to reduce the graph size quickly so that the in-memory phase becomes feasible.

Solution sketch:

1. Run a fixed number of Borůvka steps (say t steps) reading edges sequentially from disk and maintaining the current component ids in memory (using union-find).
2. Each step reduces components dramatically; after t steps the contracted graph G' has much fewer vertices. Write the contracted graph to disk (one pass).
3. Load G' into memory and run an in-memory MST algorithm (Prim/Kruskal) on G'.
4. Expand contracted edges to recover the MST of the original graph.

This technique is used in external-memory and parallel MST algorithms because Borůvka phases are I/O-friendly (edge scans) and reduce the problem size for heavier in-memory computation.

4 Fredman–Tarjan detailed phase analysis (expanded)

High-level goal. Fredman and Tarjan (1987) combined two ideas to push MST running times very close to linear in the comparison model:

1. Use an amortized-efficient priority queue (Fibonacci heaps) to implement Prim-like growth cheaply (many `DECREASE-KEY` in $O(1)$ amortized).

2. Keep each local Prim-growth heap *small* by stopping growth once the heap reaches a controlled size k. Then "contract" the grown regions, repeat on the contracted graph, and iterate through phases.

This tradeoff (small heaps $\Rightarrow$ cheap `EXTRACT-MIN`s, but more phases / contractions) is tuned to make each phase cost $O(m)$ while shrinking the vertex set quickly; the number of phases is bounded by a slowly-growing function $\beta(m, n)$, yielding overall $O(m\beta(m, n))$.

Notation. Let the input graph have n vertices and m edges. We assume distinct edge weights (tie-breaking by index if needed) so the MST is unique; this is standard and does not affect asymptotic runtime.

4.0.1 Definition: the $\beta(m, n)$ function

Define $\log^{(1)} n = \log n$, $\log^{(2)} n = \log(\log n)$, and in general $\log^{(i)} n$ is the logarithm iterated i times. Then

$$\beta(m, n) \;=\; \min\{\, i \geq 1 \;:\; \log^{(i)} n \leq \tfrac{m}{n} \,\}.$$

Intuitively, $\beta(m, n)$ is the number of times you must iterate the logarithm on n until it becomes at most the average degree m/n. Note that $\beta(m, n) \leq \log^* n$ (the iterated-logarithm), and for many dense graphs $\beta(m, n)$ is a very small constant. (Examples below.)

4.0.2 Algorithm schema (phase-oriented)

The algorithm runs in *phases*. At the start of a phase the graph has t vertices (initially $t = n$). In one phase we:

1. Choose a parameter k (depending on m and t; choice explained below).

2. Mark all vertices as *unmarked*.

3. Repeat: pick an arbitrary unmarked vertex r, run a Prim-like growth (DJP/Jarník–Prim) starting at r using a Fibonacci-heap for frontier edges, but **stop** the growth as soon as either:

   - the growing tree reaches (links to) a previously *marked* vertex, or
   - the heap size (number of distinct frontier vertices stored) reaches k.

   When the growth stops, mark all vertices that were added to the grown region; record the set of edges chosen as part of the partial MST.

4. After every unmarked vertex has been processed this way, contract each grown region into a single super-vertex; remove self-loops and keep parallel edges (with their weights).

5. Proceed to the next phase on the contracted graph (now with t' vertices).

4.0.3 Why bounded heap-size helps

When Prim runs on an ordinary graph up to size-k heaps, the cost of the expensive `EXTRACT-MIN` operations is only $O(\log k)$ each instead of $O(\log n)$. Using Fibonacci heaps gives `DECREASE-KEY` in $O(1)$ amortized, so across the whole phase:

- Every edge is considered for decrease-key at most a constant number of times $\Rightarrow$ total cost from `DECREASE-KEY` is $O(m)$.
- The number of `EXTRACT-MIN` operations in the phase equals the number of root-of-region extractions. If there are t starting vertices (super-vertices) in the phase, we perform at most t such meaningful `EXTRACT-MIN`s. Each `EXTRACT-MIN` costs $O(\log k)$. Thus the phase cost from extractions is $O(t \log k)$.

So the total work in a phase is

$$O\big(m + t \log k\big).$$

4.0.4 Choosing k to make one phase cost $O(m)$

We set k so that $t \log k = O(m)$. A convenient (and standard) choice is

$$\log k = \frac{2m}{t} \quad \Longrightarrow \quad k = 2^{2m/t}.$$

With this choice $t \log k = 2m$, and the phase cost becomes $O(m)$. (Note: other constant factors lead to the same asymptotic reasoning.)

4.0.5 How much contraction occurs in one phase?

At the end of a phase, every contracted super-vertex (a grown region) *owns* at least k incident edges (one or both endpoints counted) except possibly a small exceptional set. More precisely, because each grown region was grown until it either hit a marked vertex or its heap reached size k, we can show each super-vertex is responsible for at least $k/2$ distinct incident edges on average (details in classic lecture notes). Therefore the number t' of vertices after contraction satisfies

$$t' \;\le\; \frac{2m}{k}.$$

Plugging our choice $k = 2^{2m/t}$ gives a contraction factor that is (roughly) at least exponential in $2m/t$; intuitively t drops very fast.

4.0.6 Number of phases (relation to $\beta(m, n)$)

We start with $t_0 = n$. After one phase,

$$t_1 \le \frac{2m}{k_0} \quad \text{where } k_0 = 2^{2m/t_0}.$$

Unwinding the recurrence and using the definition of iterated logarithms yields that the algorithm finishes in at most

$$\beta(m, n) \;=\; \min\{i \ge 1 : \log^{(i)} n \le 2m/n\}$$

phases (up to constant adjustments inside the logs). Thus the total time is at most

$$\underbrace{(\text{phases})}_{\le \beta(m,n)} \times \quad \underbrace{O(m)}_{\text{cost per phase}} \quad = \; O\big(m\,\beta(m, n)\big).$$

4.0.7 Concrete examples for $\beta(m, n)$

- If $m \geq c\, n \log n$ (say $m = n \log n$), then $\frac{m}{n} = \log n$ and already $\log^{(1)} n \leq m/n$. Hence $\beta(m, n) = 1$ and the algorithm is linear $O(m)$.
- If $m = \Theta(n)$ (sparse graph), then $m/n = \Theta(1)$. Iterating log on n until it becomes $O(1)$ takes about $\log^* n$ iterations, so $\beta(m, n) = \Theta(\log^* n)$. The running time is $O(m \log^* n)$, i.e. nearly linear.
- In general $\beta(m, n) \leq \log^* n$, and for many practical graphs $\beta(m, n)$ is a very small constant (often 1 or 2).

4.0.8 Summary and remarks

- Fredman–Tarjan exploit Fibonacci heaps and bounded-size Prim growth to make every phase cost $O(m)$ and make the number of phases small (bounded by $\beta(m, n)$), thus achieving $O(m\beta(m, n))$.
- Later refinements reduced the multiplicative β factor: Gabow–Galil–Spencer–Tarjan gave $O(m \log \beta(m, n))$, Chazelle and others used soft-heaps / advanced data structures to reach bounds involving the inverse-Ackermann function $\alpha(\cdot)$, and Pettie–Ramachandran eventually matched decision-tree complexity. See references below.

4.0.9 Pseudocode (phase-level)

FredmanTarjanMST(G)
let current graph $G_0 \leftarrow G$, $t \leftarrow n$
while $t > 1$ **do**
    choose $k \leftarrow 2^{2m/t}$
    mark all vertices in G_0 as unmarked
    for each unmarked vertex r in G_0 **do**
        grow a Prim-like tree T from r using an F-heap,
  stopping when (T reaches a marked vertex) OR (heap size $\geq k$)
        mark all vertices of T
        emit edges of T as confirmed MST edges
    end for
    contract each marked-grown region into a vertex; remove self-loops
    update G_0, set $t \leftarrow$ number of super-vertices
end while
return collected MST edges =0

4.0.10 Where the $\beta(m, n)$ shows up in analysis

Two steps produce the multiplicative β factor in the simple Fredman–Tarjan analysis:

1. We make each phase cost $O(m)$ by choosing k so that $t \log k = O(m)$.
2. The number of phases is at most the number of times iterated-log must be applied to n until it is $\leq m/n$; by definition this count is $\beta(m, n)$.

Hence the $O(m\beta(m, n))$ bound.

Practical note.

Although the asymptotic improvement is interesting, Fibonacci heaps (pointer-heavy) are not always the fastest in practice; many implementations use pairing heaps or radix-based integer heaps depending on weight representation. Also, randomized algorithms (Karger–Klein–Tarjan) and other deterministic refinements give linear or near-linear algorithms under reasonable models.