# Lecture 5: Linear Sorting

## Review

- Comparison search lower bound: any decision tree with $n$ nodes has height $\geq \lceil \lg(n+1) \rceil - 1$
- Can do faster using random access indexing: an operation with linear branching factor!
- **Direct access array** is fast, but may use a lot of space ($\Theta(u)$)
- Solve space problem by mapping (**hashing**) key space $u$ down to $m = \Theta(n)$
- **Hash tables** give **expected** $O(1)$ time operations, **amortized** if dynamic
- Expectation input-independent: choose hash function randomly from **universal** hash family
- Data structure overview!
- Last time we achieved faster find. Can we also achieve faster sort?

| Data Structure | Operations $O(\cdot)$ | | | | |
|---|---|---|---|---|---|
| | Container | Static | Dynamic | Order | |
| | `build(X)` | `find(k)` | `insert(x)` `delete(k)` | `find_min()` `find_max()` | `find_prev(k)` `find_next(k)` |
| Array | $n$ | $n$ | $n$ | $n$ | $n$ |
| Sorted Array | $n \log n$ | $\log n$ | $n$ | $1$ | $\log n$ |
| Direct Access Array | $u$ | $1$ | $1$ | $u$ | $u$ |
| Hash Table | $n_{(e)}$ | $1_{(e)}$ | $1_{(a)(e)}$ | $n$ | $n$ |

## Comparison Sort Lower Bound

- Comparison model implies that algorithm decision tree is binary (constant branching factor)
- Requires # leaves $L \geq$ # possible outputs
- Tree height lower bounded by $\Omega(\log L)$, so worst-case running time is $\Omega(\log L)$
- To sort array of $n$ elements, # outputs is $n!$ permutations
- Thus height lower bounded by $\log(n!) \geq \log((n/2)^{n/2}) = \Omega(n \log n)$
- So merge sort is optimal in comparison model
- Can we exploit a direct access array to sort faster?

---

## Direct Access Array Sort

- **Example:** [5, 2, 7, 0, 4]
- Suppose all keys are **unique** non-negative integers in range $\{0, \ldots, u-1\}$, so $n \leq u$
- Insert each item into a direct access array with size $u$ in $\Theta(n)$
- Return items in order they appear in direct access array in $\Theta(u)$
- Running time is $\Theta(u)$, which is $\Theta(n)$ if $u = \Theta(n)$. Yay!

```
def direct_access_sort(A):
    "Sort A assuming items have distinct non-negative keys"
    u = 1 + max([x.key for x in A])    # O(n) find maximum key
    D = [None] * u                     # O(u) direct access array
    for x in A:                        # O(n) insert items
        D[x.key] = x
    i = 0
    for key in range(u):               # O(u) read out items in order
        if D[key] is not None:
            A[i] = D[key]
            i += 1
```

---

- What if keys are in larger range, like $u = \Omega(n^2) < n^2$?
- **Idea!** Represent each key $k$ by tuple $(a, b)$ where $k = an + b$ and $0 \leq b < n$
- Specifically $a = \lfloor k/n \rfloor < n$ and $b = (k \bmod n)$ (just a 2-digit base-$n$ number!)
- This is a built-in Python operation `(a, b) = divmod(k, n)`
- **Example:** [17, 3, 24, 22, 12] $\Rightarrow$ [(3,2), (0,3), (4,4), (4,2), (2,2)] $\Rightarrow$ [32, 03, 44, 42, 22]$_{(n=5)}$
- How can we sort tuples?

## Tuple Sort

- Item keys are tuples of equal length, i.e. item $x.key = (x.k_1, x.k_2, x.k_2, \ldots)$.
- Want to sort on all entries **lexicographically**, so first key $k_1$ is most significant
- How to sort? **Idea!** Use other **auxiliary sorting algorithms** to separately sort each key
- (Like sorting rows in a spreadsheet by multiple columns)
- What order to sort them in? Least significant to most significant!
- **Exercise:** [3**2**, 0**3**, 4**4**, 4**2**, 2**2**] $\implies$ [**4**2, **2**2, **3**2, **0**3, **4**4] $\implies$ [03, 22, 32, 42, 44]$_{(n=5)}$

---

- **Idea!** Use tuple sort with **auxiliary direct access array sort** to sort tuples $(a, b)$.
- **Problem!** Many integers could have the same $a$ or $b$ value, even if input keys distinct
- Need sort allowing **repeated keys** which preserves input order
- Want sort to be **stable**: repeated keys appear in output in same order as input
- Direct access array sort cannot even sort arrays having repeated keys!
- Can we modify direct access array sort to admit multiple keys in a way that is stable?

## Counting Sort

- Instead of storing a single item at each array index, store a chain, just like hashing!
- For stability, chain data structure should remember the order in which items were added
- Use a **sequence** data structure which maintains insertion order
- To insert item $x$, `insert_last` to end of the chain at index $x.key$
- Then to sort, read through all chains in sequence order, returning items one by one

```python
def counting_sort(A):
    "Sort A assuming items have non-negative keys"
    u = 1 + max([x.key for x in A])    # O(n) find maximum key
    D = [[] for i in range(u)]         # O(u) direct access array of chains
    for x in A:                        # O(n) insert into chain at x.key
        D[x.key].append(x)
    i = 0
    for chain in D:                    # O(u) read out items in order
        for x in chain:
            A[i] = x
            i += 1
```

## Radix Sort

- **Idea!** If $u < n^2$, use tuple sort with **auxiliary counting sort** to sort tuples $(a, b)$
- Sort least significant key $b$, then most significant key $a$
- Stability ensures previous sorts stay sorted
- Running time for this algorithm is $O(2n) = O(n)$. Yay!
- If every key $< n^c$ for some positive $c = \log_n(u)$, every key has at most $c$ digits base $n$
- A $c$-digit number can be written as a $c$-element tuple in $O(c)$ time
- We sort each of the $c$ base-$n$ digits in $O(n)$ time
- So tuple sort with **auxiliary counting sort** runs in $O(cn)$ time in total
- If $c$ is constant, so each key is $\leq n^c$, this sort is linear $O(n)$!

```python
def radix_sort(A):
    "Sort A assuming items have non-negative keys"
    n = len(A)
    u = 1 + max([x.key for x in A])              # O(n) find maximum key
    c = 1 + (u.bit_length() // n.bit_length())
    class Obj: pass
    D = [Obj() for a in A]
    for i in range(n):                           # O(nc) make digit tuples
        D[i].digits = []
        D[i].item = A[i]
        high = A[i].key
        for j in range(c):                       # O(c) make digit tuple
            high, low = divmod(high, n)
            D[i].digits.append(low)
    for i in range(c):                           # O(nc) sort each digit
        for j in range(n):                       # O(n) assign key i to tuples
            D[j].key = D[j].digits[i]
        counting_sort(D)                         # O(n) sort on digit i
    for i in range(n):                           # O(n) output to A
        A[i] = D[i].item
```

| Algorithm | Time $O(\cdot)$ | In-place? | Stable? | Comments |
|---|---|---|---|---|
| Insertion Sort | $n^2$ | $Y$ | $Y$ | $O(nk)$ for $k$-proximate |
| Selection Sort | $n^2$ | $Y$ | $N$ | $O(n)$ swaps |
| Merge Sort | $n \log n$ | $N$ | $Y$ | stable, optimal comparison |
| Counting Sort | $n + u$ | $N$ | $Y$ | $O(n)$ when $u = O(n)$ |
| Radix Sort | $n + n \log_n(u)$ | $N$ | $Y$ | $O(n)$ when $u = O(n^c)$ |