CMPT231

Lecture 4: ch8, 11

Linear-time Sort and Hash Tables

Psalm 90:10-12 (ESV)

The **years of our life** are seventy,
or even by reason of strength eighty;

yet their span is but **toil** and **trouble**;
they are soon gone, and we **fly away**.

Who considers the power of your **anger**,
and your **wrath** according to the **fear** of you?

So teach us to **number our days**
that we may get a **heart of wisdom**.

Outline for today

- Proving all **comparison** sorts are l (n lg n)
- **Linear-time** non-comparison sorts:
  - **Counting** sort
  - **Radix** sort and analysis
  - **Bucket** sort and probabilistic proof
- **Hash tables**:
  - Collision handling by **chaining**
  - Hash **functions** and **universal** hashing
  - Collision handling by **open addressing**

Summary of sorting algorithms

- **Comparison** sorts (ch2, 6, 7):
  - **Insertion** sort: V (n^2), easy to program, slow
  - **Merge** sort: V (n lg n), out-of-place copy (slow)
  - **Heap** sort: V (n lg n), in-place, max-heap
  - **Quicksort**: V (n^2) worst-case, V (n lg n) average
    - and small (fast) constant factors
- **Linear**-time non-comparison sorts (ch8):
  - **Counting** sort: k distinct values: V (k)
  - **Radix** sort: d digits, k values: V (d(n+k))
  - **Bucket** sort: uniform distribution: V (n)

Comparison sorts are Ω(n lg n)

- **Decision tree** model of computation:
  - **Leaves** are possible outputs
    - i.e., **permutations** of the input
  - **Nodes** are decision points
    - i.e., when **comparisons** are made
  - A **path** through the tree is one run of algorithm
- Num **leaves** = num **permutations** = n!
- Num **comparisons** = num **nodes** along path
- So **worst-case** complexity = **depth** of tree:
  - = l (lg(num leaves)) = l (lg(n!))
  - = l (n lg n) (by **Stirling**) .

Outline for today

- Proving all comparison sorts are l (n lg n)
- **Linear-time non-comparison sorts:**
  - **Counting sort**
  - **Radix sort and analysis**
  - Bucket sort and probabilistic proof
- Hash tables:
  - Collision handling by chaining
  - Hash functions and universal hashing
  - Collision handling by open addressing

Linear-time sorts

- Ways to **beat** the V (n lg n) barrier
- By using **assumptions** on input:
  - e.g., known range or distribution of values
  - e.g., **numeric** values we can perform **arithmetic** on
- But linear-time sorts not always **worth** it
- For real-world arrays, V (n) and V (n lg n) are very similar
  - Up to $n = 10^6$, $\lg n < 21$, a smallish factor
- For realistic n, a fast n lg n sort like **Quicksort** may have smaller constants than a linear-time sort
  - But **recursion** is expensive (function calls)

Hybrid algorithms

- (#7.4-5) : QuickSort + Insertion sort
  - One pass with **Quicksort**, stop when length < c
  - Second pass with **insertion** sort
    - Items shift at most c positions over
- TimSort: Merge sort + Insertion sort
  - Default in **Python**, Java (<7), Android, etc
  - Take advantage of monotone **runs** in real data
  - Use **run stack** to track merges and exploit **cache locality**
  - Merge with minimal extra **memory** or **copying**
  - **Stable**, best-case O(n), worst O(n lg n)

Counting sort

- **Assume**: values are **integers** in {0, …, k}
- **Out-of-place** sort:
  - **Census** array (size k) tallies a histogram
  - Items copied into **output** array
- **Stable**: preserves order of duplicates
- **Complexity**: V (n+k) (watch out if k gets too big!)

```
def counting_sort(A, n, k):
  out[ 1 .. n ]

 # new output array
  census[ 0 .. k ]
      # new array for census
  for j in 1 .. n:
      # take census
    census[ A[ j ] ]++
  for i in 1 .. k:
      # cumulative sums
```

Radix sort

- (How **IBM** made its fortune! Punch cards, ca 1900)
- **Assume**: values have at most d digits
- Sort one **digit** at a time, **least**-significant first
  - **MSD** using recursion (with call overhead)
- Use a **stable** sort, e.g., **counting** sort (**why**?)

```
def radix_sort( A, n, d ):
  for i in 1 .. d:
    stable_sort( A on digit i )
```

3	7	4	5
2	9	1	3
1	0	1	6
2	0	1	6
-	9	1	3

Radix sort: complexity

- **Input**: n items of d digits, each with k values (e.g., $k=10$)
- e.g., using **counting** sort as the stable sort:
  - d iterations, each $V(n+k)$
  - So total complexity is $V(d(n+k))$
- **Digits** need not be base $k=10$!
  - Smaller **base** $k \Rightarrow$ more **iterations** d
  - Fewer **digits** $d \Rightarrow$ each **counting** sort $V(n+k)$ takes longer

Radix: choosing digit size

- b-bit items can be **split** into r-bit digits:
  - Then $d = \frac{b}{r}$ **digits**, each with $k = 2^r - 1$ **values**
  - e.g., b = 32-bit items in r = 8-bit digits ⇒ d = 4, k = 255
- **Choose** r = lg n: then $\Theta\left(\left(\frac{b}{r}\right)(n + 2^r)\right)$ $= \Theta\left(\left(\frac{b}{\lg n}\right)(2n)\right) = \Theta\left(\frac{bn}{\lg n}\right)$
- e.g., to sort n = 2^{16} integers of b = 64-bits:
  - ⇒ Use r = 16-bit digits

Outline for today

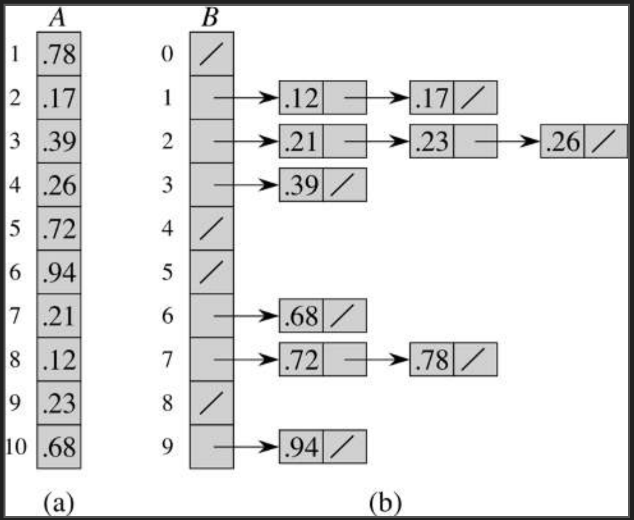
- Proving all comparison sorts are l (n lg n)
- Linear-time non-comparison sorts:
  - Counting sort
  - Radix sort and analysis
  - **Bucket sort and probabilistic proof**
- Hash tables:
  - Collision handling by chaining
  - Hash functions and universal hashing
  - Collision handling by open addressing

Bucket sort

Assume: values **uniformly** distributed over [0,1)

(For range [a, b), use linear transform to [0, 1))

- Divide *[0,1)* into *n* equal **buckets**
  - Can use **array**, or **linked list**, etc.
- **Distribute** input into buckets: *V (n)*
- **Sort** each bucket (e.g., **insertion** sort)



Bucket sort: complexity

- Let n_i be number of **items** in the i-th bucket
  - Sorting a bucket with **insertion** sort takes $O(n_i^2)$
- **Intuition**: uniform distribution $\Rightarrow n_i \approx 1$
- **Expected** time of bucket sort: $E[T(n)]$

$$= E\left[\Theta(n) + \sum O(n_i^2)\right]$$

$$= \Theta(n) + O\left(\sum E[n_i^2]\right)$$ (linearity of expectation)

$$= \Theta(n) + O\left(\sum\left(2 - \frac{1}{n}\right)\right)$$ (by lemma)

$= \Theta(n) + O(2n - 1)$ = O(n)

Lemma: $\underline{E\left[n_i^2\right]}$ = 2 - 1/n

- Use **indicator var**: X_{ij} = 1 iff j-th **item** falls in i-th **bucket**
- Number of **items** in i-th bucket is $n_i = \sum_{j=0}^{n-1} X_{ij}$
- So $E\left[n_i^2\right] = E\left[\left(\sum_{j=0}^{n-1} X_{ij}\right)^2\right]$

$$= \sum_{j=0}^{n-1} E\left[X_{ij}^2\right] + 2\sum_{j=0}^{n-1}\sum_{k=0}^{j-1} E[X_{ij}X_{ik}]$$

- Think of these as entries in a j-k **matrix**
  - Consider **diagonal** and **off-diagonal** terms separately:

Lemma, continued

- $E\left[n_i^2\right] = \sum_{j=0}^{n-1} E\left[X_{ij}^2\right] + 2\sum_{j=0}^{n-1}\sum_{k=0}^{j-1} E[X_{ij}X_{ik}]$
- For **diagonal** terms:
  $E\left[X_{ij}^2\right] = 0^2 P(X_{ij} = 0) + 1^2 P(X_{ij} = 1) \; = 1^2\left(\frac{1}{n}\right) = \frac{1}{n}$
- For **off-diagonal** terms: items j $\neq$ k are **independent**,
  so $E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] \; = \left(\frac{1}{n}\right)\left(\frac{1}{n}\right) = \frac{1}{n^2}$

Lemma, QED

- So $E\left[n_i^2\right] = \sum_{j=0}^{n-1} E\left[X_{ij}^2\right] + 2\sum_{j=0}^{n-1}\sum_{k=0}^{j-1} E[X_{ij}X_{ik}]$

$$= \sum_{j=0}^{n-1}\left(\frac{1}{n}\right) + 2\sum_{j=0}^{n-1}\sum_{k=0}^{j-1}\left(\frac{1}{n^2}\right)$$

$$= (n)\left(\frac{1}{n}\right) + 2\left(\frac{n(n-1)}{2}\right)\left(\frac{1}{n^2}\right) = 2 - \frac{1}{n}$$

- This proves the **lemma**, proving **bucket sort** is V (n)
- **Assumptions**: input values **uniformly** distributed

Outline for today

- Proving all comparison sorts are l (n lg n)
- Linear-time non-comparison sorts:
  - Counting sort
  - Radix sort and analysis
  - Bucket sort and probabilistic proof
- **Hash tables:**
  - **Collision handling by chaining**
  - Hash functions and universal hashing
  - Collision handling by open addressing

Hash tables

- **Dictionary** of key-value pairs, with this **interface**:
  - `insert(T, k, x)`: **add** item x with key k
  - `search(T, k)`: **find** an item with key k
  - `delete(T, x)`: **remove** specific item x
- Better than regular array (**direct addressing**) when:
  - **Range** of possible keys too huge to allocate
  - Actual keys are only **sparse** subset of possible keys
- e.g., only have items at keys 0, 2, 40201300:
  - Direct addressing would allocate 40,201,300 entries!

Hashing

- Hash **function** $h(k) : U \to \mathbb{Z}_m$ (i.e., {0, …, m-1}) maps from **universe** U of possible keys to a set of m **buckets**
  - Use h(k) as **key** instead of k
- Hash **collision**: two keys, **same** bucket: $h(k_i) = h(k_j)$
  - A good hash function should **minimise** collisions
  - Various collision **handling** methods
    - Let's start with **chaining** via linked lists
- Similar to **bucket sort**, but
  - Hash function maps unknown **distribution** of keys in U to **uniform** distribution on buckets

Hash table operations

- Assuming collision handling via **linked lists**:
- `insert(T, k, x)`:
  - insert x at **head** of list at bucket h(k)
  - O(1) complexity; assumes x **not already** in list
- `search(T, k)`:
  - **linear search** every item in bucket h(k)
  - $O(n_{h(k)})$, where $n_{h(k)}$ = **num items** in bucket h(k)
- `delete(T, x)`:
  - if arg is a **pointer** directly to item, then O(1)
  - if arg is a **key**, then need to **search** for it first: $O(n_{h(k)})$

Load factor

- Search **efficiency** depends on how **full** buckets are: $n_{h(k)}$
- **Load factor** v = n/m:
  - n = total number of **items** stored in hash table
  - m = number of **buckets**
  - v is **average** num items per bucket: $E\left[n_{h(k)}\right]$
- **Unsuccessful** search takes average V (1+v)
  - Computing the **hash function** takes V (1)
  - **Linear search** goes through entire bucket
  - Expected **length** of bucket’s linked list is v
- **Successful** search is also V (1+v):

Hash table search: Θ(1+α)

- Num **items** searched = **position** of x in linked list at h(k)
  - = Number of **collisions** after x was inserted
- Use an **indicator**: X_{ij} = 1 iff $h(k_i) = h(k_j)$
  - P(i and j collide) = $E[X_{ij}]$ = 1/m
- Expected num **items** searched:

$$= E\left[\left(\frac{1}{n}\right)\sum_i (\text{num items})\right]$$

$$= E\left[\left(\frac{1}{n}\right)\sum_i \left(1 + \sum_j X_{ij}\right)\right]$$ (number of collisions)

Successful search is Θ(1+α)

$$= \left(\frac{1}{n}\right)\sum_i \left(1 + \sum_j E[X_{ij}]\right)$$ (linearity of expectation)

$$= \left(\frac{1}{n}\right)\sum_i \left(1 + \sum_j \left(\frac{1}{m}\right)\right)$$ (probability of collision)

$$= \left(\frac{1}{n}\right)\sum_i 1 + \left(\frac{1}{nm}\right)\sum_i \sum_j 1$$ (independent of i, j)

$$= 1 + \left(\frac{1}{nm}\right)\left(\frac{n(n-1)}{2}\right) = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$$

Outline for today

- Proving all comparison sorts are $\Omega(n \lg n)$
- Linear-time non-comparison sorts:
  - Counting sort
  - Radix sort and analysis
  - Bucket sort and probabilistic proof
- Hash tables:
  - Collision handling by chaining
  - **Hash functions and universal hashing**
  - Collision handling by open addressing

Hash functions

- Assume $U = \mathbb{N} = \{1, 2, 3, \ldots\}$
  - i.e., keys can be converted to **natural numbers**
  - e.g., strings encoded using **ASCII** or UTF-8
- Want h(k) **uniformly** distributed on $\mathbb{Z}_m$
  - But distribution of keys k is **unknown**
  - Also, keys k_i and k_j might not be **independent**
- Various hashing **strategies**:
  - **Division** hash
  - **Multiplication** hash
  - **Universal** hashing

Division hash

- h(k) = k mod m
  - **Simplest** function mapping $\mathbb{N} \to \mathbb{Z}_m$
- **Fast** to compute: if $m = 2^p$ (i.e., a power of 2), this is just selecting the p **least-significant** bits
- But: if k is a **string** using a radix-2^p representation, then **permuting** the string gives **same** hash (#11.3-3)
- So try m **prime** and not too close to a power of 2

Multiplication hash

- h(k) = floor(m(kA mod 1)) (choose **constant** 0 < A < 1)
  - **Multiply** k∗A → take **fractional** part
  - → **multiply** by m → **round** down
- Fast **implementation** using $m = 2^p$:
  - Let w be the native machine **word size** (num bits)
  - **Choose** a w-bit integer s (0 < s < 2^w) and let A = $\frac{s}{2^w}$
  - **Multiply** s∗k: product has 2w bits in two words r_0, r_1
  - **Select** the p most-significant bits of lower word

Universal hashing

- For any **fixed** choice of hash function, can always find **bad input** resulting in lots of hash **collisions**
- Why not **randomly** select from a **pool** H of hash functions?
- Want pool to have **universal hash** property:
  - For any two keys $j \neq k$, at most $|H|/m$ hash functions in H cause a **collision**: $h(j) = h(k)$
  - i.e., $P(h(j) = h(k)) \leq 1/m$
- Then expected **bucket size** is still $O(1+v)$
  - So average complexity of **search** is still $O(1)$

Outline for today

- Proving all comparison sorts are l (n lg n)
- Linear-time non-comparison sorts:
  - Counting sort
  - Radix sort and analysis
  - Bucket sort and probabilistic proof
- Hash tables:
  - Collision handling by chaining
  - Hash functions and universal hashing
  - **Collision handling by open addressing**

Open addressing

- Another method of **collision** handling:
  - Store keys **directly** in table, no linked lists
- Hash **function** $h : U \times \mathbb{Z}_m \to \mathbb{Z}_m$
  - A **probe sequence** is h(k,0), h(k,1), h(k,2), …
- To **insert** an item in table, first try h(k,0)
  - If already **occupied**, try next in sequence: h(k,1)
  - Will eventually try **all** slots (full **coverage**) if probe sequence is a **permutation** of $\mathbb{Z}_m$
- **Search** is similar: check if found desired **key**
- Hash table will still **overflow** if n > m

Probe sequencing

- Choose a hash function that gives us **uniform hashing**:
  - Each of the m! **permutations** of $\mathbb{Z}_m$ is equally likely to be the **probe sequence** for a given key
- **Linear** probing: h(k,i) = h(k) + i
  - Try h(k), then h(k)+1, etc. (modulo m)
  - But: long **runs** get **longer** (more likely to hit)
- **Quadratic** probing: h(k,i) = $h(k) + c_1 i + c_2 i^2$
  - Must choose c_1, c_2 to ensure full **coverage**
  - But: **collision** on initial h(k) $\Rightarrow$ full **sequence collision**

Double hashing

- Use **two** hash functions: h(k, i) = $h_1(k) + ih_2(k)$
  - Try $h_1(k)$ **first**, then use h_2 to **jump** around
- For full **coverage**, ensure m and $h_2(k)$ are **relatively prime**
  - i.e., no common **factors** other than 1
  - e.g., let m = 2^p and ensure $h_2(k)$ always **odd**
  - e.g., let m be **prime**, and ensure 1 < $h_2(k)$ < m
- Each **combination** of $h_1(k)$ and $h_2(k)$ yields a **different** probe sequence:
  - Total **number** of sequences is $\Theta(n^2)$

Outline for today

- Proving all **comparison** sorts are l (n lg n)
- **Linear-time** non-comparison sorts:
  - **Counting** sort
  - **Radix** sort and analysis
  - **Bucket** sort and probabilistic proof
- **Hash tables**:
  - Collision handling by **chaining**
  - Hash **functions** and **universal** hashing
  - Collision handling by **open addressing**

Visualisations of sorting

- The Sound of Sorting (YouTube playlist)
- Visualgo: interactive demos:
  - sorting, binary heaps, hash tables, etc.
- Toptal: Comparison of sort algorithms
- Mike Bostock's "Visualizing Algorithms":
  - Fisher-Yates shuffle, sorting