CMPT231

Lecture 3: ch5-6

Heaps, Queues, and Quicksort

James 1:19-21 (NASB)

This you know, my beloved brethren:
but everyone must be **quick to hear**,
slow to speak and **slow to anger**;

for the **anger of man**
does not achieve the **righteousness of God**.

Therefore, putting aside all **filthiness**
and all that remains of **wickedness**,
in **humility** receive the **word implanted**,
which is able to **save your souls**.

Outline for today

- **Heap** sort (ch5)
  - Intro to **trees** (more in ch12)
  - Binary heaps and **max-heaps**
  - **Heap** sort
  - Max-heaps for **priority queue**
- **Quicksort** (ch6)
  - Lomuto **partitioning** and **complexity** analysis
  - **Randomised** Quicksort and analysis
- Monte-Carlo **matrix multiply** checking

Summary of sorting algorithms

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

Binary trees

- **Graph**: collection of nodes and edges
  - Edges may be directed or undirected
- **Tree**: directed acyclic graph (DAG)
  - One node designated **root**
  - **Parent**: immediate neighbour toward root
  - **Leaf**: node with no children
  - **Degree**: maximum number of children per node
  - **Height** of node: max num edges to leaf descendant
  - **Depth** of node: num edges to root
  - **Level**: all nodes of same depth
- **Binary tree**: degree 2

Outline for today

- Heap sort (ch5)
  - Intro to trees (more in ch12)
  - **Binary heaps and max-heaps**
  - **Heap sort**
  - Max-heaps for priority queue
- Quicksort (ch6)
  - Lomuto partitioning and complexity analysis
  - Randomised Quicksort and analysis
- Monte-Carlo matrix multiply checking

Binary heaps

- **Array** storage for certain types of **binary trees**:
  - Put node i's two **children** at 2i and 2i+1
  - **Fill** tree left-to-right, one level at a time
  - e.g.: `[ 2, 8, 4, 7, 5, 3, 1, 6 ]`
- The **max-heap** property: every node's value is ≤ its parent
  - (min-heap: ≥)
- `max_heapify()` (O(lg n)):
  - move a node i to satisfy **max-heap property**
- `build_max_heap()` (O(n)):
  - turn an unordered array into a max-heap
- `heapsort()` (O(n lg n)): in-place **sort**

max_heapify() on single node

- **Input**: binary heap A and node index i
  - **Precondition**: left and right sub-trees of i are each separate max-heaps
- **Postcondition**: entire subtree at i is a max-heap
- Algorithm:
  1. Find **largest** of: i, **left** child of i, or **right** child of i
  2. If i is **not** the largest, then:
     a. **Swap** i with the largest
     b. **Recurse** (or iterate) on that subtree

max_heapify()

```
def max_heapify( A, i ):
  max = i
  if 2i <= length(A) and A[ 2i ] > A[ max ]:
    # left
    max = 2i
  else if 2i+1 <= length(A) and A[ 2i+1 ] > A[ max ]:   # right
    max = 2i+1
  if max != i:
    swap( A[ i ], A[ max ] )
    max_heapify( A, max )


      # recurse
```

- **Try** it on previous heap at i=1
- **Running time**?

Building a max-heap

- **Input**: array A, in any order
  - **Postcondition**: A is a max-heap
- Algorithm:
  1. **Last half** of array is all **leaves**
  2. Run `max_heapify()` on each item in **first half**
     - **Descending** order: **subtrees** are already max-heaps

```
for i = floor( length(A)/2 ) to 1:
  max_heapify( A, i )
```

Try it: `[ 5, 2, 7, 4, 8, 1 ]`

Max-heap: complexity

- **Group** iterations of `for` loop by **height** h of node:
  - Each call to `max_heapify(i)` takes O(h)
  - **Num of nodes** with height h is $\leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$
    - Reaches that bound when tree is **full**
- Total **running time** T(n): $T(n) = \sum_{h=0}^{\lg n} \left(\frac{n}{2^{h+1}} \right) O(h)$

$$\leq n \sum_{h=0}^{\infty} \left(\frac{1}{2^{h+1}} \right) O(h) \;= n \sum_{h=1}^{\infty} \left(\frac{1}{2^{h}} \right) O(h) \;= O(n)$$

- $\Rightarrow$ Can build a max-heap in **linear** time!
  - But it’s not quite a **sorting** algorithm….

Heap sort

- **Algorithm**:
  1. Make array a **max-heap**
  2. **Repeat**, working backwards from end of array:
     - **Swap** root with last leaf of heap
     - **Shrink** heap by 1 and **re-apply** `max_heapify()`
- **Loop invariant**:
  - First portion of array is still a **max-heap**
  - Last portion of array is **sorted** (largest items)
- **Complexity**: T(n) = Θ(n lg n)
  - Θ(n) calls to `max_heapify()` (Θ(lg n))
- **Try** it: `[ 5 2 7 4 8 1 ]`

Outline for today

- Heap sort (ch5)
  - Intro to trees (more in ch12)
  - Binary heaps and max-heaps
  - Heap sort
  - **Max-heaps for priority queue**
- Quicksort (ch6)
  - Lomuto partitioning and complexity analysis
  - Randomised Quicksort and analysis
- Monte-Carlo matrix multiply checking

Priority queue

- We can use binary heaps to make a **priority queue**:
  - Set of items with attached priorities
- **Interface** (set of operations) for priority queue:
  - `insert(A, item, pri)`: **add** an item
  - `find_max(A)`: **get** item with highest priority
  - `pop_max(A)`: same but also **delete** the item
  - `set_pri(A, item, pri)`: **change** (increase) priority of item
- **Initialise** queue by building a max-heap:
  - `find_max()` is easy: just return A[1]
  - `pop_max()` also easy: remove A[1] and

Insert into queue

- `set_pri()`: start at i and **bubble** up to proper place:

```
A[ i ] = pri
while i > 1 and A[ i/2 ] < A[ i ]:
```

swap(A[i/2], A[i]) i = i/2

  - **Complexity**: num iterations = $\Theta(\lg n)$
- `insert()`: make **new** node, then set its **priority**:

```
A.size++
A[ size ] = item
set_pri( A, A.size, pri )
```

  - **Complexity**: same as `set_pri()`: $\Theta(\lg n)$
- For speed, often **pre-allocate** A as fixed-length array

Priority queue operations

- **Build** queue (with max-heap): $\Theta(n)$
- **Fetch** highest priority item: $\Theta(1)$
- Fetch and **remove** highest priority item: $\Theta(\lg n)$
- **Change** priority of an item: $\Theta(\lg n)$
- **Insert** new item: $\Theta(\lg n)$

Outline for today

- Heap sort (ch5)
  - Intro to trees (more in ch12)
  - Binary heaps and max-heaps
  - Heap sort
  - Max-heaps for priority queue
- **Quicksort** (ch6)
  - **Lomuto partitioning and complexity analysis**
  - Randomised Quicksort and analysis
- Monte-Carlo matrix multiply checking

Randomised algorithms

- **Vegas**-style: always **correct**, fast on **average**
  - But still slow in **worst-case**
- **Monte Carlo**: always **fast**
  - But not always **correct**!
  - **Approximate**: margin of error ε
  - **Stochastic**: probability P of being correct
  - Estimate improves with more **computation** time (iterations)

Quicksort

- **Divide**: partition A[lo .. hi] such that:
  - max(A[lo .. piv-1]) ≤ A[piv] ≤ min(A[piv+1 .. hi])
  - Not always **balanced**; this is the "magic sauce"
- **Conquer**: recurse on each part:
  - quicksort(A, lo, piv-1) and quicksort(A, piv+1, hi)
  - No **combine** / merge step needed
- **In-place** sort (only uses swaps) (unlike merge sort)
- **Worst** case still $\Theta(n^2)$, but
  - **Average** case is Θ(n lg n), with small constants
- One of the **best** sorts for **arbitrary** inputs
  - When we can only use **comparisons**

Partitioning (Lomuto)

- One option: pick last item as the **pivot**
- **Walk** through array from left to right:
  - Throw items **smaller** than pivot to left part of array
  - Items **larger** than pivot stay in right part of array
- Lastly, **swap** pivot in-between two parts

```
def partition( A, lo, hi ):
  piv = A[ hi ]

     # Lomuto partition
  split = lo


# Where to send items <= piv
  for cur = lo to hi-1:
    if A[ cur ] <= piv:
     # Send small items away
```

Quicksort: complexity

- **Worst** case: every partition is maximally **uneven**:
  - **Pivot** is either largest or smallest in subarray
  - T(n) = T(n-1) + T(0) + Θ(n) = $\Theta(n^2)$
  - Example **inputs** that do this?
- **Best** case: every partition is exactly **half**:
  - T(n) = 2T(n/2) + Θ(n) = Θ(n lg n)
  - Example **inputs** that give this?
- What about **average** case, assuming random input?

Average case complexity

- **Intuition**: on average, get splits in-between best and worst
  - If say, average split is 90% vs 10%, then:
  - T(n) = T(0.90n) + T(*0.10n) + *Θ(n)
  - Still results in O(n lg n)
- If assume splits **alternate** between best and worst:
  - Only adds O(n) work to each of O(lg n) levels
  - Still O(n lg n)! (But maybe larger constants)

Quicksort with constant splits

- (p.178 #7.2-5): All splits are v vs 1-v, with 0 < v < 1/2
  - ⇒ what is min/max **depth** of leaf in recursion tree?
- **Min depth**: follow smaller (v) side of each split
  - How many splits m until reach **leaf** (1 item array)? $\alpha^m n = 1 \Rightarrow m = -\frac{\log(n)}{\log(\alpha)}$
- **Max depth**: same with 1-v side: $-\frac{\log(n)}{\log(1-\alpha)}$
- Both are Θ(log n)
- ⇒ with **constant-ratio** splits, complexity still Θ(n lg n)

Quicksort with median split

- **Best** case splits happen when pivot is **median**:
  - Half of items smaller, half of items larger
  - Not same as average when distribution is **skewed**
- **Median** (rank finding) algorithm in O(n): see ch9
  - **Partitioning** also takes only O(n), so
  - **Quicksort** $T(n) = 2T(n/2) + O(n) = O(n \lg n)$ (always!)
- But, in practise:
  - **Extra** work, splits are usually **already** good
  - Benchmarks **slower** than merge sort

Outline for today

- Heap sort (ch5)
  - Intro to trees (more in ch12)
  - Binary heaps and max-heaps
  - Heap sort
  - Max-heaps for priority queue
- Quicksort (ch6)
  - Lomuto partitioning and complexity analysis
  - **Randomised Quicksort and analysis**
- Monte-Carlo matrix multiply checking

Randomised Quicksort

- With **random** input, get nice $\Theta(n \lg n)$ behaviour
- But **presorted** input gives **worst** case behaviour
  - Much **real-world** data is at least partially presorted
  - Choice of **last** element (hi) as pivot
- Great candidate for **randomised** algorithm:
  - Before partitioning, **swap** hi with a random item
- Still **possible** to get worst-case behaviour, but **unlikely**
  - **Vegas**-style: always **correct**, and usually **fast**

```
def rand_partition( A, lo, hi ):
  swap( A[ hi ], A[ random( lo, hi ) ] )
  partition( A, lo, hi )
```

R-Quicksort: complexity

- Assume all items **distinct**
- Name items according to **true** order: $\{z_i\}_{i=1}^{n}$
- Analyse complexity by counting **comparisons** performed
  - **Worst** case: compare all pairs (z_i, z_j) : $\Theta(n^2)$
- No comparison can happen **multiple** times, because
  - Comparisons are only done against **pivots**, and
  - Each pivot is used only **once** and not **revisited**
- So what is the **probability** of a pair (z_i, z_j) being compared?

R-Quicksort: pair comparison

- A pair (z_i, z_j) is **compared** only if:
  - Either z_i or z_j is chosen as a pivot **before** any other item ordered in-between them: $\{z_i, z_{i+1}, \ldots, z_{j-1}, z_j\}$
- Otherwise z_i and z_j would be on **opposite** sides of a split, and would **never** be compared
- **Probability** of this happening: $2\left(\frac{1}{j-i+1}\right)$

R-Quicksort: total comparisons

Sum over all pairs (z_i, z_j):

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \text{ (let k=j-i)}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

$$= \sum_{i=1}^{n-1} O(\ln n) \text{ (harmonic series)}$$

$$= O(n \lg n)$$

Outline for today

- Heap sort (ch5)
  - Intro to trees (more in ch12)
  - Binary heaps and max-heaps
  - Heap sort
  - Max-heaps for priority queue
- Quicksort (ch6)
  - Lomuto partitioning and complexity analysis
  - Randomised Quicksort and analysis
- **Monte-Carlo matrix multiply checking**

Randomised mat-mul check

- Recall **matrix multiply**: naive $\Theta(n^3)$, Strassen $\Theta(n^{2.81})$
  - Best-known: Coppersmith-Winograd, $\Theta(n^{2.376})$
- What if we have 3 n x n matrices A, B, C:
  - **Check** if A ∗ B = C, faster than full multiply?
- **Frievald**'s matrix-multiply checker in $\Theta(n^2)$:
  - If A ∗ B = C, always returns True (0% false-negatives)
  - If A ∗ B ≠ C, returns False > 50% of the time
- If returns True, run it k times:
  - False-positive rate < 2^{-k}, in time $O(kn^2)$

Frievald's algorithm

- Make a random **boolean** vector $\vec{r} = \{r_i\}_1^n$:
  - $P(r_i = 1)$ = 0.5 for all i, independently
  - i.e., flip a fair coin n times
- **Return value**: check if $A \cdot (B \cdot \vec{r}) = C \cdot \vec{r}$
  - Each **multiply** is only a (n x n) matrix by a (n x 1) vector
  - $\Rightarrow$ total time still only $\Theta(n^2)$
- Example of a **Monte-Carlo** style algorithm
- If A ∗ B = C, this always returns True
- If A ∗ B ≠ C, want $P(A \cdot (B \cdot \vec{r}) \neq C \cdot \vec{r})$ > 0.5

Frievald: false-positive rate

- Let D = AB - C: by assumption, D $\neq$ 0, so choose $d_{ij} \neq 0$
  - $\Rightarrow$ Want to **show** $P(D\vec{r} = 0) \leq 0.5$
- $D\vec{r}$ is 0 **iff** all its elts are 0, so $P(D\vec{r} = 0) \leq P((D\vec{r})_i = 0)$
- This is a **dot product**: $(D\vec{r})_i = \sum_{k=1}^{n} d_{ik} r_k = d_{ij} r_j + y$
- **Two** possibilities: if y = 0: $P((D\vec{r})_i = 0) \; = P(d_{ij} r_j = 0) = P(r_j = 0) = 0.5$
- If y $\neq$ 0, then $P((D\vec{r})_i = 0) \; = P(r_j = 1 \text{ and } d_{ij} = -y) \leq P(r_j = 1) = 0.5$
- In **either** case, $P((D\vec{r})_i = 0) \leq 0.5$

Outline for today

- **Heap** sort (ch5)
  - Intro to **trees** (more in ch12)
  - Binary heaps and **max-heaps**
  - **Heap** sort
  - Max-heaps for **priority queue**
- **Quicksort** (ch6)
  - Lomuto **partitioning** and **complexity** analysis
  - **Randomised** Quicksort and analysis
- Monte-Carlo **matrix multiply** checking