Divide-and-conquer and the Master Theorem

William Hendrix

Lecture 12

Outline

- Review
  - Bit vectors, Heaps, Union-find
- Union-Find
- Prefix and suffix trees: not on exams
- Divide-and-conquer
- The Master Theorem

Data structures

- Bit vectors
  - Dictionary w/ O(1) worst-case search, insert, and delete time
  - Disadvantages: integers only, space depends on data range
- Priority Queue
  - Specialized to find the optimum value in a set (max or min)
  - Implemented with heap
  - **Max()**: constant time for max-heap
  - **DeleteMax()**: swap max with last leaf and call PercolateDown(1): $O(\lg n)$
  - **Insert(x)**: add as last leaf and PercolateUp(n): $O(\lg n)$
  - **Heapify()**: call PercolateDown(i) from end to beginning: $O(n)$
- Union-Find
  - Represent partition of a dataset
  - Implemented by array
  - **Find(x)**: return partition ID for *x*
  - **Union(a, b)**: join partitions containing *a* and *b* together

Union-Find implementation

- Array contains element IDs
- Partition IDs are elements pointing to themselves
- Initially, all elements are isolated:

0	1	2	3	4	5	6	7
{0}	{1}	{2}	{3}	{4}	{5}	{6}	{7}

- Find(x)
  - Follow links until you hit a partition ID
  - Return partition ID
  - *O(n)* time

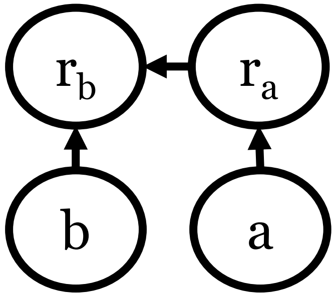
- Union(a, b)
  - Point Find(a) to b
  - *O(n)* time

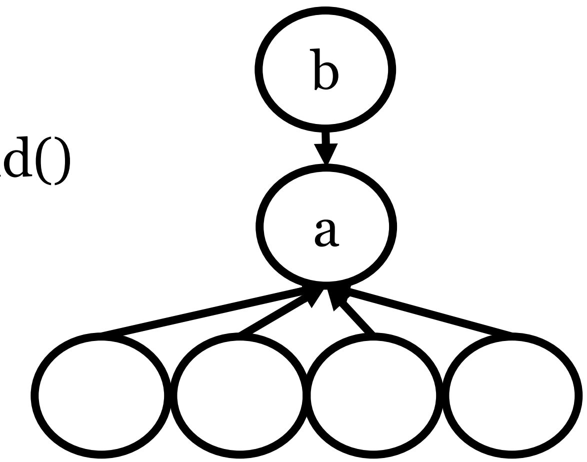
```
Algorithm: Find(x)
if unionfind[x] = x then
    return x;
else
    id = Find(unionfind[x]))
    return id;
end
```

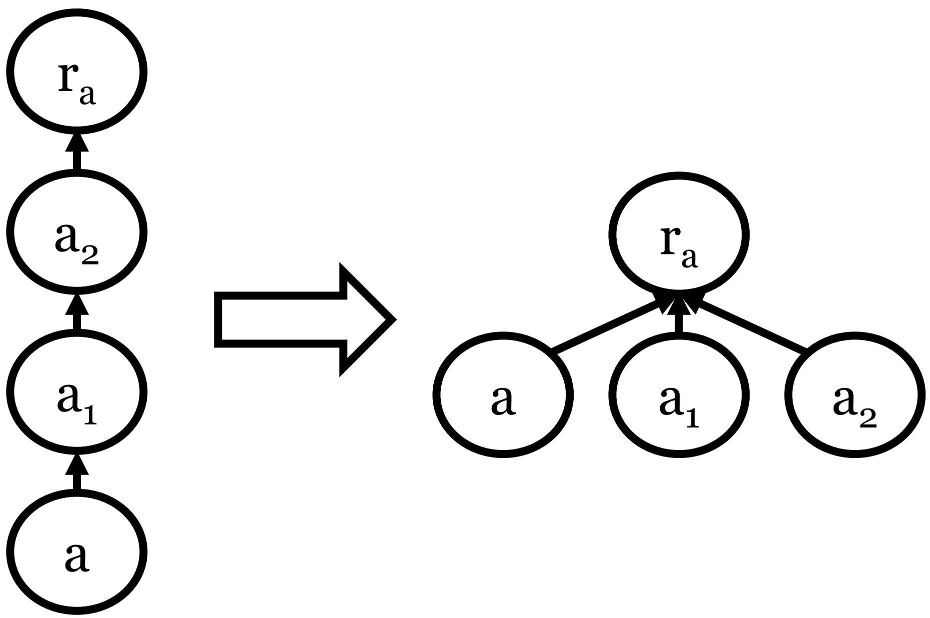
```
Algorithm: Union(a, b)
id = Find(a);
unionfind[id] = b;
```

Optimizing Union-Find

- Union complexity depends on Find
- Find complexity depends on height of tree
- First idea: add Find(a) to Find(b) (or vice versa)
- Second idea: add the smaller tree to the larger
  - Swap a and b if a > b (or a < b)
  - Min (max) value always ends up as root
- Third idea: flatten structure when we call Find()







Union-Find operations

- **Find(x)**
  - Recursively point to answer
  - $O(\alpha(n))$
  - Generally less than 5 for conceivable *n*
- **Union(a, b)**
  - Call Find on both sides first
  - Always point to max (min)
  - $O(\alpha(n))$
- The Ackermann function

$$\alpha^{-1}(n) = \overbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}^{n+1 \text{ twos}} - 3$$

$$\alpha^{-1}(1) = 2^2 - 3 = 1$$
$$\alpha^{-1}(2) = 2^4 - 3 = 13$$
$$\alpha^{-1}(3) = 2^{16} - 3 = 65,533$$
$$\alpha^{-1}(4) = 2^{65536} - 3 \ldots \text{is very big}$$

Algorithm: Find(x)

if unionfind$[x] \neq x$ **then**
  id = Find(unionfind$[x]$);
  unionfind$[x] = id$;
end
return unionfind$[x]$;

Algorithm: Union(a, b)

ra = Find(a);
rb = Find(b);
if $ra > rb$ **then**
  Swap ra and rb;
end
unionfind$[ra] = rb$;

Algorithm strategies

- Exhaustive search
  - Try everything!
- Greedy algorithm
  - Always pick the best option
- New strategy: organize data
  - List out values needed to solve the current problem
  - Enumerate all data operations
  - Choose a data structure based on common operations
  - May even use multiple data structures for same data
    - E.g., hash table to remove duplicates and heap to find min
  - May introduce metadata to further accelerate computation
    - Data about data
    - E.g., store every 100^{th} value in sorted array in another array
  - Pay small up-front cost to organize, save order of complexity later
  - Useful in conjunction with any other strategy

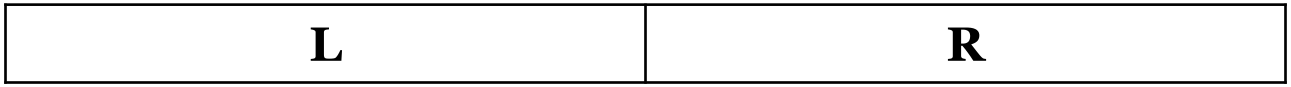
Divide-and-conquer

- Another algorithm strategy

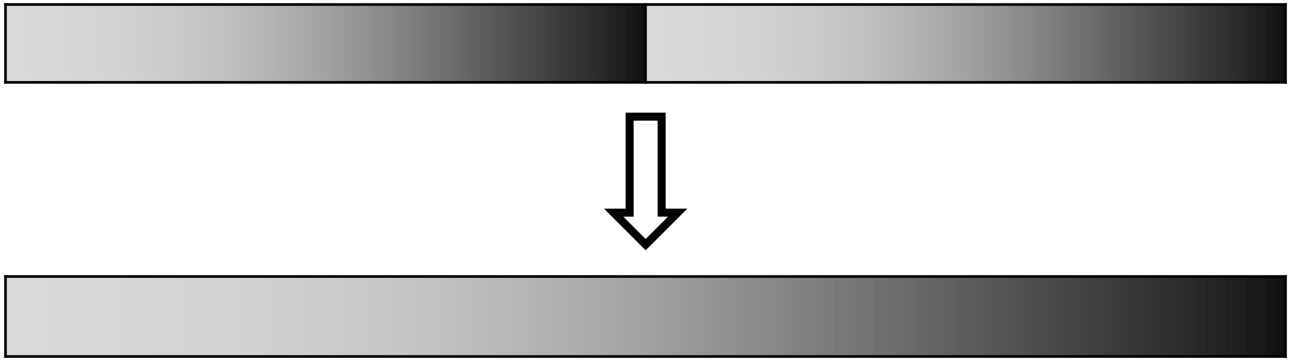
- **Goal**
  - Reduce complexity of high-complexity algorithms
- **Outline**
  - Divide large problems into one or more subproblems of roughly the same size
    - E.g., split array into 2 halves, 3 thirds, etc.
  - Solve subproblems via recursion
  - Combine solutions to subproblems into solution for full problem
  - Solve small problems directly (*base case*)
- **Intuition**
  - If combining solutions is easier than solving directly, divide-and-conquer solution will be faster

Divide-and-conquer example

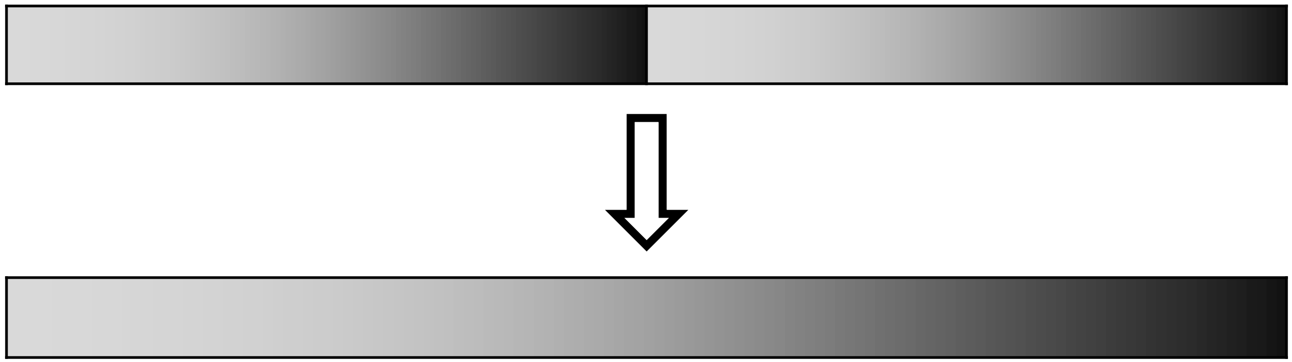
- Sorting
  - Several $O(n^2)$ algorithms
- Applying divide-and-conquer
  - Split array into two halves



  - Sort half-arrays recursively
    - Base case: one element
  - Combine two sorted half-arrays into one sorted array



Divide-and-conquer example



- Fill in result left-to-right
- Min is min(left) or min(right)
- 2^{nd} value is next value of selected half or min(unselected half)
- Continue until both arrays have emptied into result
  - After one array is empty, just add the other
- Time to combine: $O(n)$
  - Better than $O(n^2)$!

- Algorithm: MergeSort

MergeSort

Input: $data$: the data to sort (must be comparable)
Input: n: the number of elements in data
Output: a permutation of $data$ such that $data[1] \leq \ldots \leq data[n]$
Algorithm: MergeSort

```
if n ≤ 1 then
    return data;
else
    mid = floor((n + 1)/2);
    left = MergeSort(data[1..mid], mid);
    right = MergeSort(data[mid + 1..n], n − mid);
    temp = array(n);
    ℓ = r = s = 1;
    while ℓ ≤ mid and r ≤ n − mid do
        if left[ℓ] < right[r] then
            temp[s] = left[ℓ];
            ℓ = ℓ + 1;
        else
            temp[s] = right[r];
            r = r + 1;
        end
        s = s + 1;
    end
    rem = mid − ℓ;
    temp[s + 1..s + rem] = left[ℓ + 1..mid];
    temp[s + rem + 1..n] = right[r + 1..n − mid];
    return temp;
end
```

MergeSort analysis

- $T(n)$: time to sort array of size n
- Sorting a large array:
  - Divide array into two halves $\Theta(1)$
  - Sort left half $T(n/2)$
  - Sort right half $T(n/2)$
  - Merge two halves $\Theta(n)$
- Total time: $T(n) = 2T(n/2) + \Theta(n)$

- Solving the recurrence:
  - $c(x)$ = ???
  - Not $T(n) = T(n-1) + \ldots + T(n-k) + f(n)$
- *We need another technique!*

Analysis: divide-and-conquer

- In general, we need 3 factors to determine divide-and-conquer complexity:

  $T(n) = aT(n/b) + f(n)$

  – b : number of pieces we are dividing problem into

  – a : number of recursive calls (often $a = b$)

  – $f(n)$: time required to combine subproblem solutions

- **Master Theorem**

  – Gives complexity for $T(n)$ based on a, b, and $f(n)$

  1. Calculate $c = log_b(a)$
  2. Compare complexity of $f(n)$ to n^c
     - If $f(n) = \Theta(n^c)$, $T(n) = \Theta(f(n) \lg n)$
     - Otherwise, if $f(n)$ is *strictly smaller* than $O(n^c)$, $T(n) = \Theta(n^c)$
       – $f(n) = O(n^{c+e})$, for some $e > 0$
     - Otherwise, if $f(n) = \Omega(n^{c+e})$ and f is *regular*, $T(n) = \Theta(f(n))$
       – Strictly more than n^c
       – Regular: $af(n/b) < f(n)$, for large n
       – All functions that grow faster than linear are regular

Formal statement

Master Theorem. If T is an increasing function that satisfies the recurrence

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b \geq 1$, then:

$$T(n) = \begin{cases} \Theta(n^c), & \text{if } f(n) = O(n^{c-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(n^c \lg n), & \text{if } f(n) = \Theta(n^c) \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{c+\epsilon}) \text{ for some } \epsilon > 0 \\ & \quad \underline{\text{and}}\ af(n/b) < f(n) \text{ for large } n \end{cases}$$

Almost: $T(n) = \Theta(n^c + f(n))$ unless n^c and $f(n)$ are same size

For purposes of the Master Theorem, you may ignore floor and ceiling

$$\begin{aligned} \text{E.g., } T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \\ &= 2T(n/2) + O(n) \end{aligned}$$

Warning: cases 1+3 must be polynomially different (<u>not log</u>)

Application: MergeSort complexity

Master Theorem. If $T(n) = aT(n/b) + f(n)$,

$$T(n) = \begin{cases} \Theta(n^c), & \text{if } f(n) = O(n^{c-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(n^c \lg n), & \text{if } f(n) = \Theta(n^c) \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{c+\epsilon}) \text{ for some } \epsilon > 0 \\ & \quad \underline{\text{and}}\ af(n/b) < f(n) \text{ for large } n \end{cases}$$

- $T(n) = 2T(n/2) + \Theta(n)$

1. Identify variables
   - $a = 2$, $b = 2$, $f(n) = \Theta(n)$
2. Calculate c
   - $c = \log_b(a) = \log_2(2) = 1$
3. Decide case
   - n^c vs. $f(n)$? $f(n) = \Theta(n^c)$
4. Report complexity (test regularity if case 3)
   - $\Theta(n^c \lg n) = \boxed{\Theta(n \lg n)}$

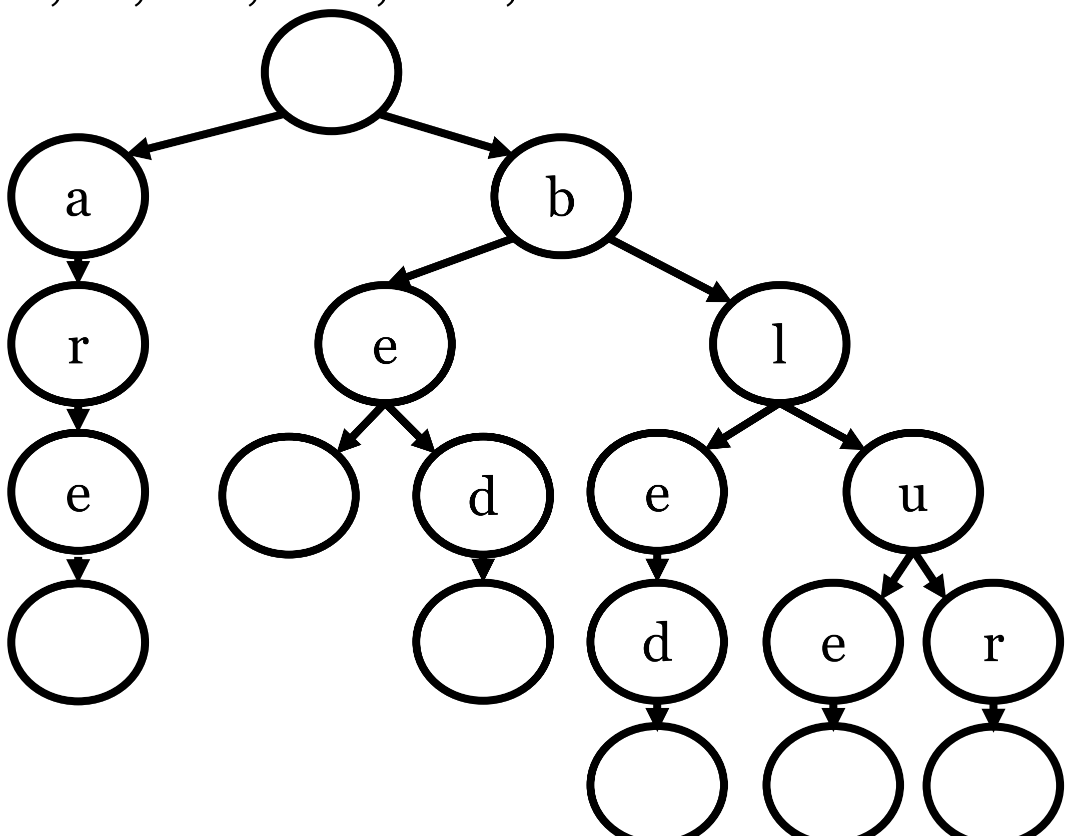
Coming up

- **Exam 2** will be next Tuesday
  - Data structures, divide-and-conquer, Master Theorem
  - **Practice Exam 2** posted on Canvas
- Exam review on Thursday
- After exam: sorting algorithms
- **Project 1** will be due Oct. 18

- **Recommended readings:** Chapter 3, Sections 4.3, 4.5, and 4.10
- **Practice problems:** 3-21 (p. 101), 4-13, 4-30, 4-32 (p. 140)

Prefix trees

- A.k.a., trie
- Nonlinear linked data structure for storing collections of *strings*
- Each node in tree represents one letter in string
- Null character represents beginning and end of word
- **Example:** are, be, bed, bled, blue, blur



Prefix tree implementation

- Linked structure
- Parent and children pointers
- Children are stored in a *map*
- Dictionary of (k, v) ordered pairs
  - **Insert(k, v)**: adds pair to map or replaces if exists (k, v_2)
  - **Search(k)**: returns v associated with k
  - **Delete(k):** removes pair associated with k
    - Complexity same as dictionary
      - Array, hash table, or BST
  - If keys are consecutive, can also be stored as array of values:

1	2	3	....	r-1	r

- *O(1)* time operations, worst-case
- May be space inefficient if range is large

Prefix tree operations

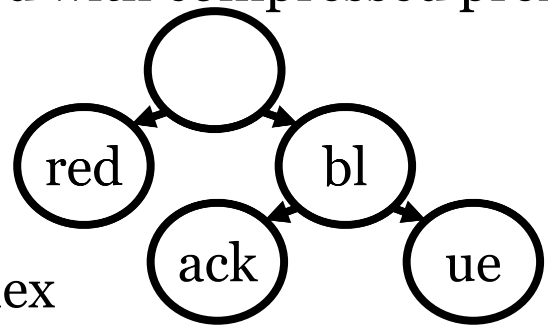
- **Contains(w)**
  - Searches for w
  - $O(|w|)$ time
- **Insert(w)**
  - Searches for w while adding nodes
  - $O(|w|)$ time
- **Delete(w)**
  - Searches for w
  - Backtracks until a parent has > 1 child, freeing nodes along the way
  - $O(|w|)$ time
- **Root()**
  - Returns root node of prefix tree
  - *O(1)* time
- **Next(c)**
  - Returns node corresponding to char *c*
  - *O(1)* time

```
Algorithm: Insert(w)
child = node.next(w[1]);
if child = NIL then
    child = NewNode();
    child.parent = node;
    node.children.Insert(w[1],
    child);
end
child.Insert(w[2..|w|]);
```

```
Algorithm: Contains(w)
if w = ∅ and node.value = '\0'
then
    return node;
else
    child = node.next(w[1]);
    if child = NIL then
        return NIL;
    else
        return
        child.Contains(w[2..|w|]);
    end
end
```

Prefix tree analysis

- Excellent for string matching applications
  - E.g., spellchecking, autocorrect, etc.
- May take large amount of space
  - Worst case: $O(w_{sum})$ nodes
    - Pointers are larger than characters…
- Can be mitigated with compressed prefix trees



  - More complex
  - Potential complexity issues if continually modifying
- **Suffix trees**
- Prefix tree containing all suffixes of a word
  - $O(|w|)$ space if compressed properly
- Excellent for greatest common substring, etc.