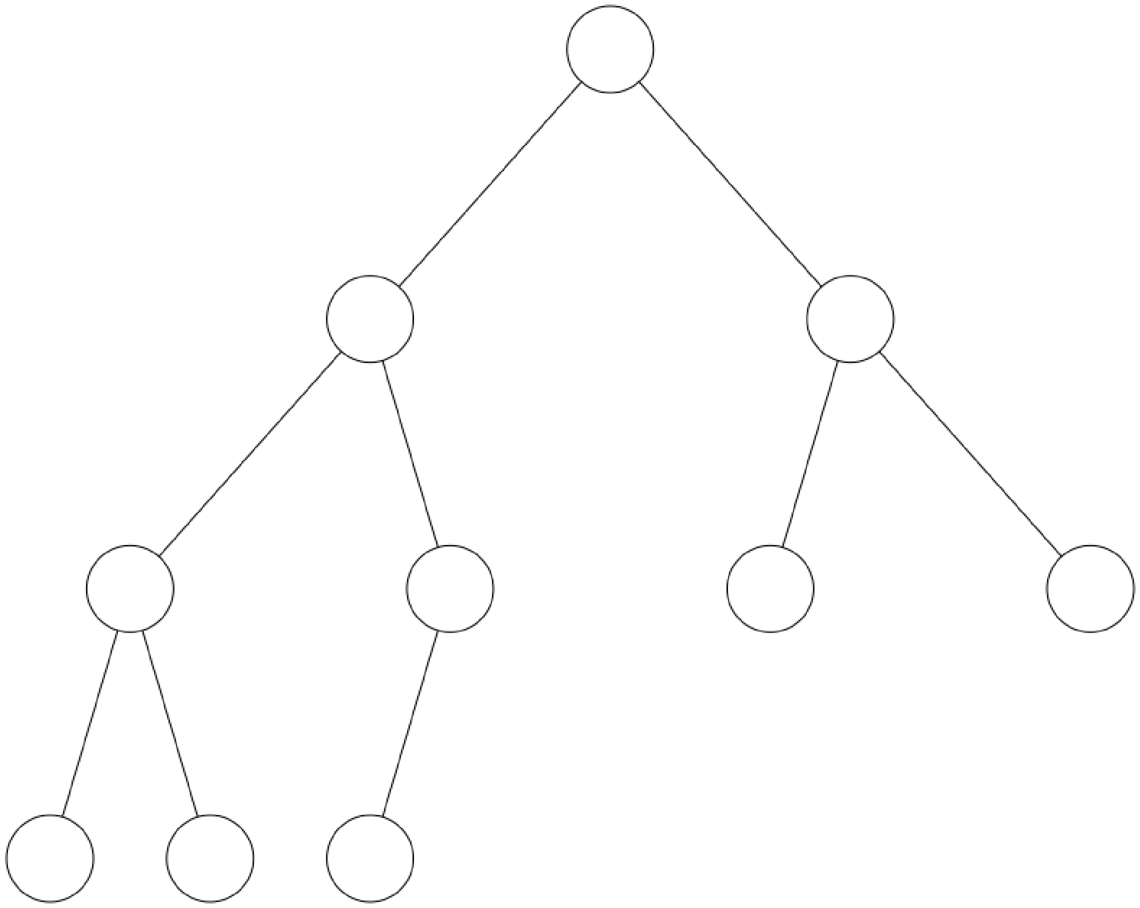
Analysis of Algorithms II

Deepayan Sarkar

Heapsort

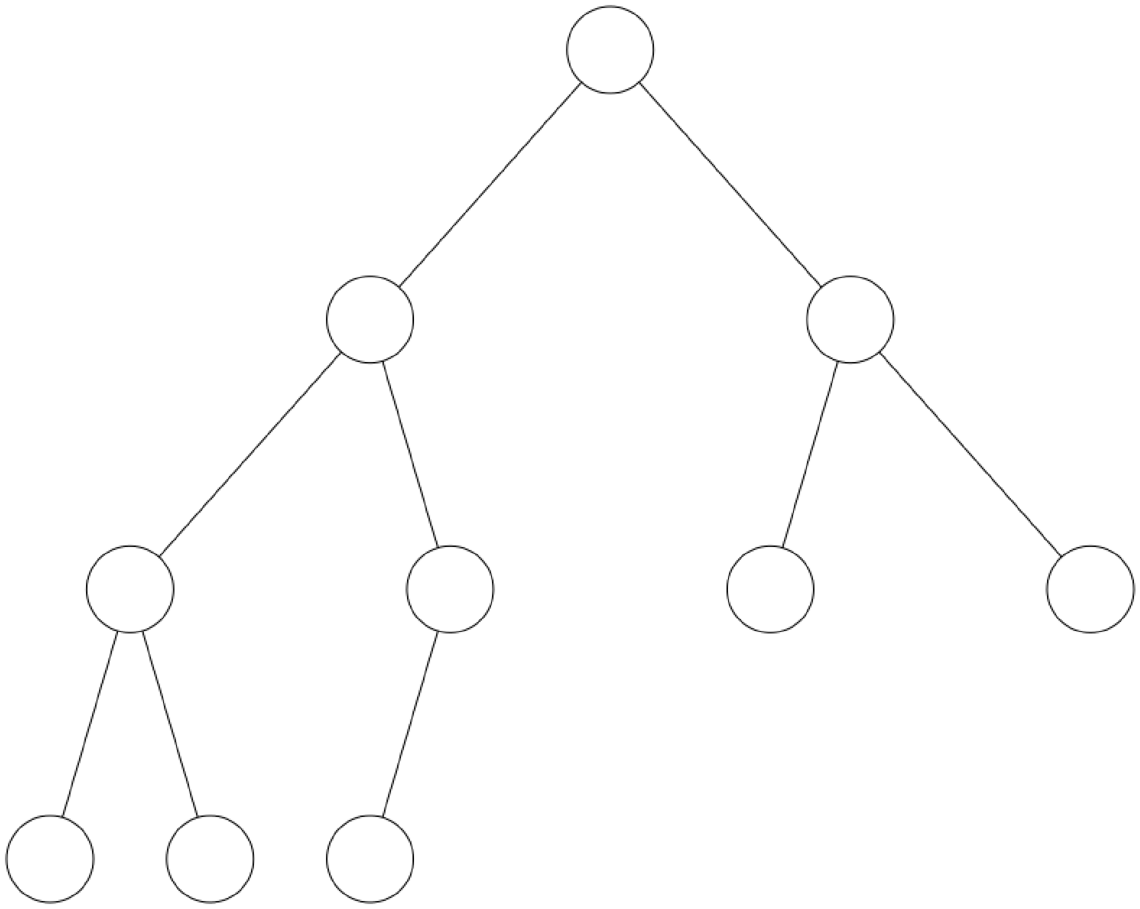
- Next we study another sorting algorithm called *heapsort*
- It has the good properties of both merge sort and insertion sort
  - It has $O(n \log_2 n)$ worst-case running time
  - It is in-place (requires only a constant amount of extra storage)
- It is based on a *data structure* known as a heap.

The abstract heap data structure



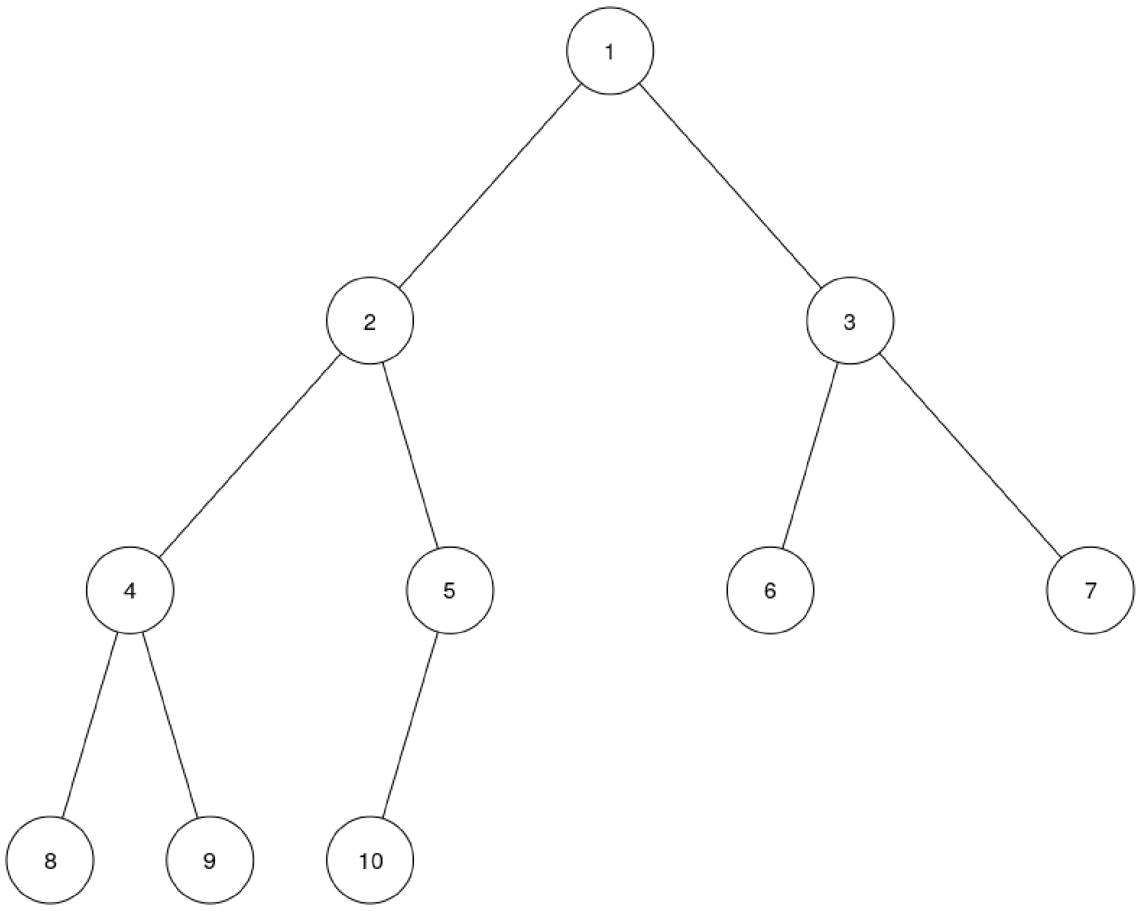
- The (binary) heap data structure is an object that we can view as a *nearly complete binary tree.*
  - Each node corresponds to an element.
  - The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.
- For each node `x`, the following operations are defined:
  - `PARENT(x)` returns the parent node
  - `LEFT(x)` returns the left child node
  - `RIGHT(x)` returns the right child node

How can we implement a heap?



- General graph $G = (V, E)$ consists of
  - V : set of vertices or nodes
  - E : set of edges
- Usually stored as list of nodes and edges / adjacency matrix
- *Trees* are a subtype of graphs
  - They have a special *root node*
  - Each node has 0 or more *child nodes*
  - Nodes with no children are called *leafs*
- Heaps are are (almost) *complete binary trees*
- This makes implementation of heaps easier than for general graphs

Implementation of a heap using arrays



- Suppose we number the nodes as shown
- Then we can define

PARENT

return floor($i/2$)

LEFT

return $2i$

RIGHT

return $2i + 1$

- Because of this, heaps are usually implemented using an array
- Specifically, a heap is an array `A` with two attributes:
  - `length(A)` gives the number of elements in the array
  - The first `heap-size(A)` elements of the array are considered part of the heap
- Note that the number of elements of an array are usually fixed
- As we will see, it is common to change the heap size in heap-based algorithms

- Index the array by $1, 2, ..., n$
- Root node has index 1
- Then as shown above, we can implement

`PARENT(i)`

return floor($i/2$)

`LEFT(i)`

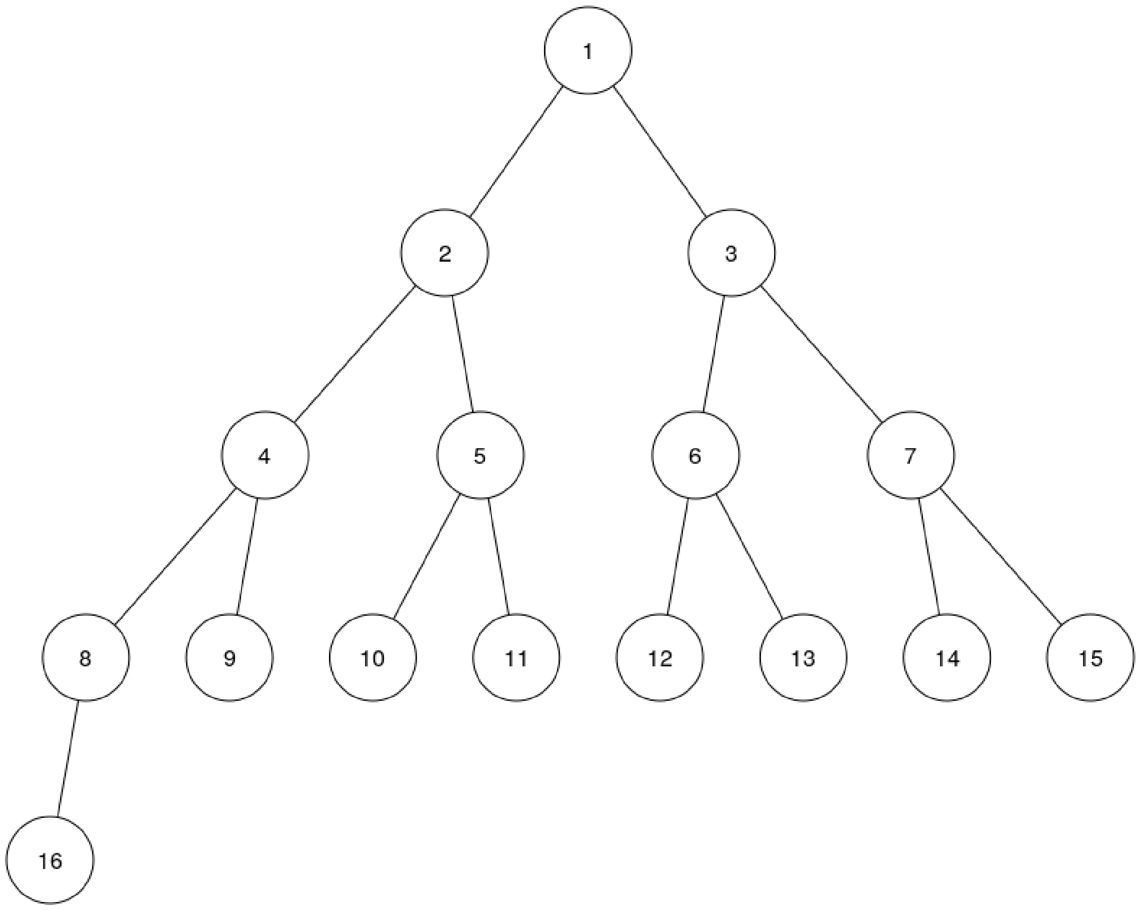
return $2i$

`RIGHT(i)`

return $2i + 1$

- In C / C++, there are shift operators `<<` and `>>` that make these efficient
- Implementations need to change if arrays are indexed from 0

Height of a heap



- View the heap as a tree

- The *height* of a node is the *number of edges* on the *longest simple downward path* from the node to a leaf.
- The *height of the heap* is the height of its root
- A heap of size n has height $\lfloor \log_2 n \rfloor$

Heap property

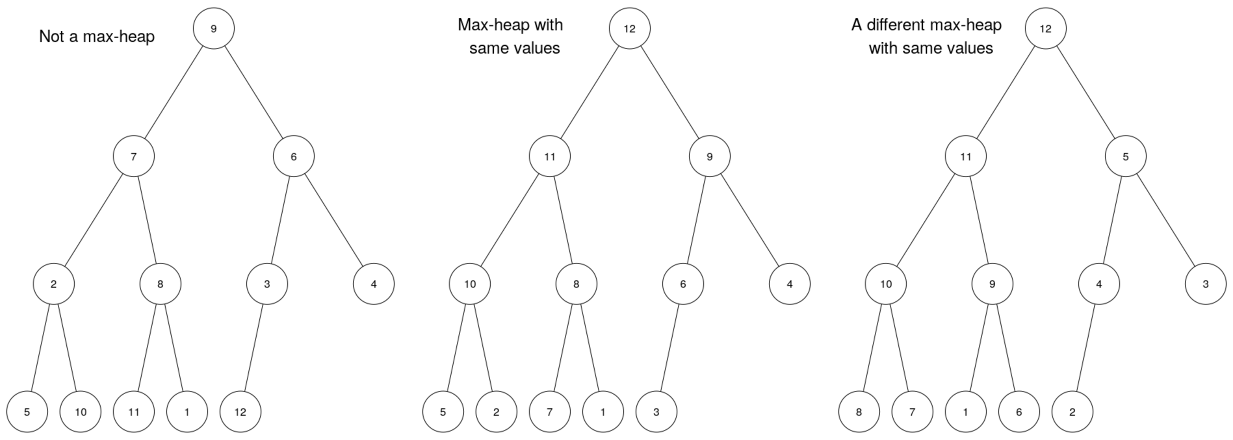
- We are usually interested in heaps that satisfy a particular property
- Depending on the property, the heap is called either a *max-heap* or a *min-heap.*
- **Max-heap**: A heap A is called a max-heap if it satisfies the "max-heap property"

$$A[PARENT(i)] \geq A[i] \text{ for all } i > 1$$

- That is, the value at every node (except the root node) is less than or equal to the value at its parent. In particular,
  - the largest element in a max-heap is stored at the root
  - The subtree rooted at any node only contains values less that or equal to the value in that node
- **Min-heap**: Similarly, a heap A is a *min-heap* if it satisfies the "min-heap property"

$$A[PARENT(i)] \leq A[i] \text{ for all } i > 1$$

Example: max-heap


Not a max-heap
9
7
6
2
8
3
4
5
10
11
1
12
Max-heap with same values
12
11
9
10
8
6
4
5
2
7
1
3
A different max-heap with same values
12
11
5
10
9
4
3
8
7
1
6
2

Algorithms for max-heaps

- For the heapsort algorithm, we will use max-heaps
- The key elements of the algorithm are
  - The `BUILD-MAX-HEAP` procedure, which produces a max-heap from an unordered input array, and
  - The `MAX-HEAPIFY` procedure, which is used to maintain the max-heap property

MAX-HEAPIFY

- Suppose that we have a heap that is almost a max-heap
- However, the max-heap property may not hold for the root element
- `MAX-HEAPIFY` fixes this error and makes it a max-heap
- The `MAX-HEAPIFY` procedure has the following inputs
  - an array A, and
  - an index i into the array
- When called, `MAX-HEAPIFY` assumes that
  - the binary trees rooted at $LEFT(i)$ and $RIGHT(i)$ are max-heaps, but
  - $A[i]$ might be smaller than its children
- `MAX-HEAPIFY` moves $A[i]$ down the max-heap so that the subtree rooted at i becomes a max-heap
- Outline: At each step,
  - The largest of the elements $A[i], A[LEFT(i)], A[RIGHT(i)]$ is determined
  - Its index is stored in the variable *largest*
- If $A[i]$ is largest, then the subtree rooted at node i is already a max-heap and the procedure terminates
- Otherwise, one of the two children has the largest element, and so
  - $A[i]$ is swapped with $A[largest]$
  - Node i and its immediate children now satisfy the max-heap property
  - But $A[largest]$ now equals the original $A[i]$, so that subtree might violate the max-heap property
  - So we call `MAX-HEAPIFY` recursively on that subtree

`MAX-HEAPIFY(A, i)`

```
l = LEFT(i)
r = RIGHT(i)
largest = i
if (l ≤ heap-size(A) and A[l] > A[i]) {
    largest = l
}
if (r ≤ heap-size(A) and A[r] > A[largest]) {
    largest = r
}
if (largest != i) {
    Swap A[i] and A[largest]
    MAX-HEAPIFY(A, largest)
}
```

Running time of MAX-HEAPIFY

- Let $T(n)$ be The running time of `MAX-HEAPIFY` for a sub-tree of size n
- Requires a constant time to compare the root with two children to decide which is largest
- If necessary, additionally requires time to `MAX-HEAPIFY` a subtree
- Claim: The size of a subtree can be at most $2n/3$.

- Proof is an exercise: Hint:
  - Height = $k = \lfloor log_2 n \rfloor$
  - Size of subtree is at most $2^k \leq 2^{\lfloor log_2 n \rfloor}$
  - Worst case when tree half-full (is that obvious?)
  - Then, $n = 2^k - 1 + 2^k/2 = 3/2 \times 2^k - 1$, and size of subtree is $m = 2^k - 1$
  - Then, $m/n = 2/3 \times \frac{1-1/L}{1-2/3L}$, where $L = 2^k$
  - The extra factor simplifies to $(3L - 3)/(3L - 2) < 1$
- This gives the recurrence

$$T(n) = T(2n/3) + \Theta(1)$$

- By the master theorem, the solution is $T(n) = O(\log_2 n)$
- We often state this by saying that runtime of `MAX-HEAPIFY` is linear in the height of the tree

Building a max-heap

- We can easily use `MAX-HEAPIFY` in a bottom-up manner to convert an array $A[1, ..., n]$ into a max-heap
- All elements $A[i]$ for $i > PARENT(n)$ are leaves of the tree, and so are already 1-element max-heaps

`BUILD-MAX-HEAP(A)`

```
heap-size(A) = length(A)
for (i = PARENT(length(A)), ..., 2, 1) {
    MAX-HEAPIFY(A, i)
}
```

To prove correctness, we can use the following loop invariant:

> At the start of each iteration of the for loop, each node $i + 1, i + 2, ..., n$ is the root of a max-heap.

Initialization

- $i = PARENT(length(A))$. All subsequent nodes are leaves so trivially max-heaps

Maintenance

- Children of any node i are numbered higher than i
- Since these are max-heaps by the loop invariant condition, it is legitimate to apply `MAX-HEAPIFY(A, i)`
- This now makes i the root of a max-heap, and the property continues to hold for all nodes numbered $> i$
- When i decreases by 1, the loop invariant becomes true for the next value of i

Termination

- At termination, $i = 0$. By the loop invariant, each node $1, 2, ..., n$ is the root of a max-heap
- In particular, this holds for node 1, the root node

Runtime of `BUILD-MAX-HEAP(A)`

- A simple upper bound for the running time is $n \log_2 n$
- Can we do better? Possibly yes, because
  - Running time for `MAX-HEAPIFY` is lower for nodes of low height
  - Such nodes are more in number
- In particular, An n-element heap has
  - Height $H = \lfloor \log_2 n \rfloor$, and
  - At height h (i.e., height $H - h$ from root node), at most 2^{H-h} nodes
- Runtime $T(n)$ of `MAX-HEAPIFY` on a node of height h is $O(h)$
- So the total run time for `BUILD-MAX-HEAP` is bounded above by

$$\sum_{h=0}^{H} 2^{H-h} O(h) = 2^H O\left(\sum_{h=0}^{H} \frac{h}{2^h}\right)$$

- Recall that

$$\sum_{k=0}^{n} kx^k < \sum_{k=0}^{\infty} kx^k = x\frac{\mathrm{d}}{\mathrm{d}x}\sum_{k=0}^{\infty} x^k = x\frac{\mathrm{d}}{\mathrm{d}x}\frac{1}{1-x} = \frac{x}{(1-x)^2}$$

- Thus we can see that

$$\sum_{h=0}^{H} \frac{h}{2^h} \leq \frac{1/2}{(1-1/2)^2} = 2$$

- As $2^H \leq n$, $T(n) = O(n)$

Heapsort

Finally, we come to the heapsort algorithm

- Use `BUILD-MAX-HEAP` to build a max-heap on the input array A of length n
- Initial heap size $s = n$
- The maximum element of the array is now stored at the root $A[1]$
- Put it into its correct final position by swapping with $A[s]$
- Now, discard this maximum element in $A[n]$ from the heap, by simply decreasing the *heap size* s by 1
- The remainder is almost a max-heap, except possibly at the root node
- Make it a max-heap by calling `MAX-HEAPIFY`
- Repeat

`HEAPSORT(A)`

```
BUILD-MAX-HEAP(A)
for (i = length(A), ..., 3, 2) {
    swap A[1] and A[i]
    heap-size(A) = heap-size(A) - 1
    MAX-HEAPIFY(A, 1)
}
```

- Exercise: Prove correctness of HEAPSORT using the following loop invariant:

  At the start of each iteration of the for loop, the subarray $A[1, ..., i]$ is a max-heap containing the i smallest elements of $A[1, ..., n]$, and the subarray $A[i+1, ..., n]$ contains the $n-i$ largest elements of $A[1, ..., n]$ in sorted order.

- Exercise: Show that runtime $T(n)$ of heapsort is

$$T(n) = O(n) + \sum_i O(\lfloor \log_2 i \rfloor) = O(n) + O\left(\sum_i \log_2 i\right) = O(n \log_2 n)$$

Probabilistic Analysis

- A common problem: finding the maximum
  - given a list of things
  - want to find the "best" among them
- Typical approach: look at each one by one, keeping track of the best
- Not much we can do to improve on this
- A variant of this problem: there is a substantial cost to updating the current 'best' value
- We can phrase this as the **hiring problem**

The hiring problem

- Suppose that your current office assistant is horribly bad, and you need to hire a new office assistant
- An employment agency sends you one candidate every day
- You interview a candidate and decide either to hire or not
- But if you don't hire the candidate immediately, you cannot hire him / her later
- You pay the employment agency a small fee to interview an applicant
- Hiring an applicant is more costly because you must also compensate the current current office assistant who you are firing

Hiring strategy: always hire the best

- You want to have the best possible person for the job at all times
- Therefore, you decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, you will fire the current office assistant and hire the new applicant
- You are willing to pay the resulting price of this strategy, but you wish to estimate what that price will be

`hire-assistant(n)`

```
best = 0 // least-qualified dummy candidate
for (i = 1, . . . , n) {
    interview candidate i
    if (i is better than best) {
        best = i
        hire candidate i
    }
}
```

- Let c_i be interview cost, and c_h be hiring cost.
- Then the total cost is $nc_i + mc_h$, where m is the number of times we hired someone new.
- The first part is fixed, so we concentrate on mc_h.

Probabilistic analysis

- Worst case:
  - we get applicants in increasing order (worst to best)
  - we hire everyone we interview
  - So $m = n$
- Best case: $m = 1$
- What is the average case?
- We need to assume a probability distribution on the input order
- Simplest model: candidates come in random order
- More precisely, their order is a uniformly random permutation of $1, 2, ..., n$
- Define

$$\begin{aligned} X_i &= \mathbf{1}\{\text{Candidate } i \text{ is hired}\} \\ X &= \sum_i X_i \end{aligned}$$

- Then $E(X_i) = 1/i \implies E(X) = \sum_{i=1}^{n} 1/i \approx \log n$
- Exercise: Can we write $E(X) = \Theta(\log n)$?
- Exercise: Determine $Var(X)$.

Quicksort

- The final general sorting algorithm we study is called quicksort
- It is among the fastest sorting algorithms in practice
- Estimating the runtime theoretically is somewhat tricky
- Quicksort is a divide-and-conquer algorithm (like merge-sort)
- The steps to sort an array $A[p, ..., r]$ are:
  - Choose an element in A as the pivot element x
  - Partition (rearrange) the array $A[p, ..., r]$ and compute index $p \le q \le r$ such that
    * Each element of $A[p, ..., q] \le x$
    * Each element of $A[q + 1, ..., r] \ge x$
    * Computing the index q is part of the partitioning procedure
  - Sort the two subarrays $A[p, ..., q]$ and $A[q + 1, ..., r]$ by recursive calls to quicksort
  - No further work needed, because the whole array is now sorted
- The procedure can thus be written as

`QUICKSORT(A, p, r)`

```
if (p < r) {
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q)
    QUICKSORT(A, q+1, r)
}
```

- The full array A of length n can be sorted with `QUICKSORT(A, 1, n)`
- Of course, the important ingredient is `PARTITION()`

Partitioning in quicksort: original version

- Quicksort was originally invented by C. A. R. Hoare in 1959
- He proposed the following `PARTITION()` algorithm

`PARTITION(A, p, r)`

```
x = A[p] // choose first element as pivot
i = p - 1
j = r + 1
while (TRUE) {
    repeat
        j = j - 1
    until (A[j] ≤ x)
    repeat
        i = i + 1
    until (A[i] ≥ x)
    if (i < j) {
        swap A[i] and A[j]
    }
    else {
        return j
    }
}
```

Correctness

- Exercise: Assuming $p < r$, show that in the algorithm above,
  - Elements outside the subarray $A[p, ..., r]$ are never accessed
  - The algorithm terminates after a finite number of steps
  - On termination, the return value j satisfies $p \leq j < r$
  - Every element of $A[p, ..., j]$ is less than or equal to every element of $A[j+1, ..., r]$

Performance of quicksort (informally)

- Runtime of `PARTITION` is clearly $\Theta(n)$ (linear)
- Worst-case: partitioning produces one subproblem with $n-1$ elements and one with 1 element

$$T(n) = T(n-1) + T(1) + \Theta(n) = T(n-1) + \Theta(n)$$

- Solved by $T(n) = \Theta(n^2)$

- Best case: always balanced split

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

- By master theorem gives $T(n) = O(n \log_2 n)$
- This happens if we can somehow ensure that the pivot is always the median
- That is of course impossible to ensure
- Average case: This turns out to be also $O(n \log_2 n)$, but the proof of this is more involved

Lomuto partitioning scheme

- We will study a slightly different version of quicksort (due to Lomuto)
- Formal runtime analysis of this version is easier

`PARTITION(A, p, r)`

```
x = A[r] // choose last element as pivot
i = p - 1
for (j = p, . . . , r-1)
    if (A[j] <= x) {
        i = i + 1
        swap(A[i], A[j])
    }
swap(A[i+1], A[r])
return i + 1
```

- This rearranges $A[p, ..., r]$ and computes index $p \leq q \leq r$ such that
  - $A[q] = x$
  - Each element of $A[p, ..., q-1] \leq x$
  - Each element of $A[q+1, ..., r] \geq x$
- The quicksort algorithm is modified as

`QUICKSORT(A, p, r)`

```
if (p < r) {
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)
}
```

Correctness of Lomuto partitioning scheme

- As the procedure runs, it partitions the array into four (possibly empty) regions.
- At the start of each iteration of the for loop in lines 3–7, the regions satisfy certain properties.
- We state these properties as a loop invariant:

  At the beginning of each iteration of the loop, for any array index k,

  1. If $p \leq k \leq i$, then $A[k] \leq x$
  2. If $i+1 \leq k \leq j-1$, then $A[k] > x$
  3. If $k = r$, then $A[k] = x$

(The values of $A[k]$ can be anything for $j \leq k < r$)

Proof of loop invariant

Initialization:

- Prior to the first iteration of the loop, $i = p - 1$ and $j = p$
- No values lie between p and i and no values lie between $i + 1$ and $j - 1$
- So, the first two conditions of the loop invariant are trivially satisfied
- The assignment $x = A[r]$ in line 1 satisfies the third condition

Maintenance:

- We have two cases, depending on the outcome of the test in line 4
- When $A[j] > x$, the only action is to increment j, after which
  - condition 2 holds for $A[j - 1]$
  - all other entries remain unchanged
- When $A[j] \leq x$, the loop increments i, swaps $A[i]$ and $A[j]$, and then increments j
- Because of the swap, we now have that $A[i] \leq x$, and condition 1 is satisfied
- Similarly, $A[j - 1] > x$, as the value swapped into $A[j - 1]$ is, by the loop invariant, greater than x

Termination:

- At termination, $j = r$
- Every entry in the array is in one of the three sets described by the invariant
- We have partitioned the values in the array into three sets:
  - those less than or equal to x
  - those greater than x
  - a singleton set containing x
- The second-last line of `PARTITION` swaps the pivot element with the leftmost element greater than x
- This moved the pivot into its correct place in the partitioned array
- The last line returns the pivot's new index

Performance of quicksort

- Again, it is easy to see that the running time of `PARTITION` is $\Theta(n)$.
- Worst case: $T(n) = \Theta(n^2)$ as before
- Best case: $T(n) = O(n \log_2 n)$ as before
- Examples of worst case:
  - Input data already sorted
  - All input values constant
- Exercise:
  - Are these worst cases for the original (Hoare) partition algorithm as well?

  - Suggest simple modifications which can “fix” these worst cases (without increasing order of runtime of `PARTITION`)

- Average case: What is the runtime of quicksort in the “average case”
- This is the expected runtime when the input order is random (uniformly over all permutations)
- A related concept: Randomized Algorithms
- An algorithm is *randomized* if it makes use of (pseudo)-random numbers
- We will analyze a randomized version of quicksort
  - This requires a “random number generator” algorithm `RANDOM(i, j)`
  - `RANDOM(i, j)` should return a random integer between i and j (inclusive) with uniform probability

Randomized quicksort

- Randomized quicksort chooses a random element as pivot (instead of the last) when partitioning

`RANDOMIZED-PARTITION(A, p, r)`

```
i = RANDOM(p,r)
swap(A[r], A[i])
return PARTITION(A, p, r)
```

- The new quicksort calls `RANDOMIZED-PARTITION` in place of `PARTITION`

`RANDOMIZED-QUICKSORT(A, p, r)`

```
if (p < r) {
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q-1)
    RANDOMIZED-QUICKSORT(A, q+1, r)
}
```

Randomized quicksort and average case

- A randomized algorithm can proceed differently on different runs with the same input
- In other words, the runtime for a given input is a random variable
- This leads to two distinct concepts:
  - Expected runtime of `RANDOMIZED-QUICKSORT` (on a given input)
  - Average case runtime of `QUICKSORT` (averaged over random input order)
- Claim: If all input elements are distinct, these two are essentially equivalent
- An alternative randomized version of quicksort is to randomly permute the input initially
- The expected runtime in that case is clearly equivalent to the average case of `QUICKSORT`
- Instead, we only choose the pivot randomly (in each partition step)
- However, this does not change the resulting partitions (as sets)
- A little thought shows that the number of comparisons is also the same
- The number of swaps may differ, but are less than the number of comparisons

Average-case analysis

- Assume that all elements of the input n-element array $A[1, ..., n]$ are distinct
- Each call to `PARTITION` has a for loop where each iteration makes one comparison ($A[j] \leq x$)
- Let X be the number of such comparisons in `PARTITION` over the *entire* execution of `QUICKSORT`
- Then the running time of `QUICKSORT` is $O(n + X)$
- This is easy to see, because
  - `PARTITION` is called at most n times (actually less)
  - In each such call, each iteration of the for loop makes one comparison contributing to X
  - The remaining operations of `PARTITION` only contribute a constant term
- To analyze runtime of quicksort, we will try to find $E(X)$
- In other words, we will not analyze contribution of each `PARTITION` call separately
- Let
  - $z_1 < z_2 < \cdots < z_n$ be the elements of A in increasing order
  - $Z_{ij} = \{z_i, ..., z_j\}$ be the set of elements between z_i and z_j, inclusive.
  - $X_{ij} = \mathbf{1}\{z_i \text{ is compared with } z_j\}$ sometime during the execution of `QUICKSORT`
- First, note that two elements may be compared at most once
  - One of the elements being compared is always the pivot
  - The pivot is never involved in subsequent recursive calls to `QUICKSORT`
- So, we can write

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

- Therefore

$$E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{ij}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(z_i \text{ is compared with } z_j)$$

- The trick to evaluating this probability is to notice that it only depends on Z_{ij}
- We want to compute

$$P(z_i \text{ is compared with } z_j)$$

- Consider the first element x in $Z_{ij} = \{z_i, ..., z_j\}$ that is chosen as a pivot (at some point)
- If $z_i < x < z_j$, then z_i and z_j will never be compared
- However, if x is either z_i or z_j, then they will be compared
- So, we want the probability that x is either z_i or z_j
- This is easy once we realize that

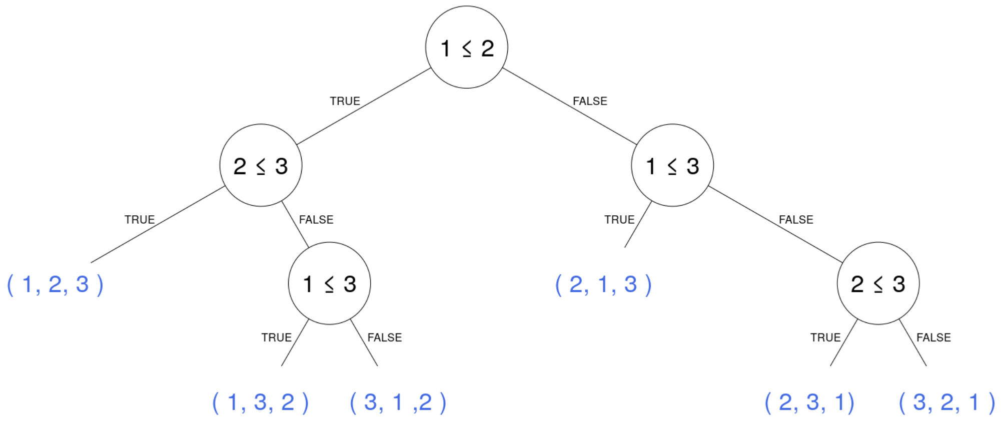
  until the first time something in Z_{ij} is chosen as a pivot, all elements in Z_{ij} remain in the *same partition* in any previous call to PARTITION (they are either all less than or greater than any previous pivot)

- Recall that pivots are chosen uniformly randomly (in `RANDOMIZED-PARTITION`)
- So any element of Z_{ij} is equally likely to be the one chosen first
- Thus the required probability is $2/|Z_{ij}| = 2/(j-i+1)$, and so

$$EX = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log_2 n) = O(n \log_2 n)$$

General lower bound for comparison-based sort

- We have now seen four different sorting algorithms
- Three of them have $O(n \log n)$ runtime
- A common property: they all use only pairwise comparison of elements to determine the result
- In other words, only ranks are important, not the actual values
- Such sorting algorithms are called *comparison sorts*
- Claim: Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case
- To see why, think of any comparison sort as a *decision tree*
  - Each comparison leads to a decision
  - A sequence of decisions leads to the correct sorted result
- For example, this is what happens when we do insertion sort on three elements a_1, a_2, a_3
- Here, $i \leq j$ denotes the act of comparing a_i and a_j



- Generally, this decision tree must be a *binary* tree (two outcomes of each comparison)
- It must have at least $n!$ leaf nodes (one or more for each possible permutation)
- Comparisons needed to reach a particular leaf: length of the path from the root node
- The worst case number of comparisons is the height of the binary tree (longest path)
- A binary tree of height h can have at most 2^h leaf nodes
- A binary tree with at least $n!$ leaf nodes must have height $h \geq \log_2 n!$
- Using Stirling's approximation $\log n! = n \log n - n + O(\log n)$,

$$h \geq \log_2(n!)/\log_2(2) = \Theta(n \log n)$$

Linear time sorting

- Sorting can be done in linear time in some special cases
- As shown above, they cannot be comparison-based algorithms
- Usually, these algorithms put restrictions on possible values
- Examples:
  - Counting sort
  - Radix sort
- Details left for a second semester project

Randomly permuting arrays

- A common requirement in randomized algorithms is to find a random permutation of an input array
- One option: assign random key values to each element, then sort the elements according to these keys

`PERMUTE-BY-SORTING(A)`

```
n = length(A)
let $P$[1,,,,n] be a new array
**for** (i = 1, . . . , n) {
    P[i] = RANDOM(1, M)
}
sort A, using P as sort keys
```

- Here M should large enough that the possibility of keys being duplicated is small
- Exercise: Show that `PERMUTE-BY-SORTING` produces a uniform random permutation of the input, assuming that all key values are distinct
- The runtime for `PERMUTE-BY-SORTING` will be $\Omega(n \log_2 n)$ if we use a comparison sort
- A better method for generating a random permutation is to permute the given array in place
- The procedure `RANDOMIZE-IN-PLACE` does so in $\Theta(n)$ time

`RANDOMIZE-IN-PLACE(A)`

```
n = length(A)
**for** (i = 1, . . . , n) {
    swap(A[i], A[ RANDOM(i, n) ])
}
```

- In the ith iteration, $A[i]$ is chosen randomly from among $A[i], A[i+1], ..., A[n]$
- Subsequent to the ith iteration, $A[i]$ is never altered.
- Procedure `RANDOMIZE-IN-PLACE` computes a uniform random permutation
- We prove this using the following loop invariant

  Just prior to the ith iteration of the for loop, for each possible $(i-1)$-permutation of the n elements, the subarray $A[1, ..., i-1]$ contains this $(i-1)$-permutation with probability $(n-i+1)!/n!$.

Initialization

- Holds trivially $(i-1=0)$
- If this is not convincing, take (just before) $i=2$ to be the initial step

Maintenance

- Assume true upto $i=1,...,k$
- Consider what happens just before $i=(k+1)$th iteration (i.e., just after kth iteration)
- Let $(X_1,X_2,...,X_k)$ be the random variable denoting the observed permutation
- For any specific k-permutation $(x_1,x_2,...,x_k)$,

$$\begin{aligned}P(X_1=x_1,X_2=x_2,...,X_k=x_k) &= P(X_k=x_k|X_1=x_1,X_2=x_2,...,X_{k-1}=x_{k-1})\\ &\quad \times P(X_1=x_1,X_2=x_2,...,X_{k-1}=x_{k-1})\\ &= \frac{1}{n-k+1}\times\frac{(n-k+1)!}{n!}=\frac{(n-k)!}{n!}\end{aligned}$$

Termination

- $i=n+1$, so each n-permutation is observed with probability $1/n!$

Further topics

- We will not discuss analysis of algorithms further
- If you are interested, an excellent book on this topic is
  Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein
- We will discuss some more algorithms in second semester projects