# Week 5: Quicksort, Lower bound, Greedy

Agenda:

- Quicksort: Average case
- Lower bound for sorting
- Greedy method

## Recall Quicksort:

- The ideas:
  - Pick one key
  - Compare to others: partition into 'smaller' and 'greater' sublists
  - Recursively sort two sublists
- Pseudocode:

```
procedure Quicksort(A, p, r)

    if p < r then
        q ← Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)


procedure Partition(A, p, r)
    ** A[r] is the key picked to do the partition

    x ← A[r]
    i ← p − 1
    for j ← p to r − 1 do
        if A[j] ≤ x then
            i ← i + 1
            exchange A[i] ↔ A[j]
    exchange A[i + 1] ↔ A[r]
    return i + 1
```

`Partition`$(A, p, r)$:

- The invariant:
  - $A[p..i]$ contains keys $\leq A[r]$
  - $A[(i+1)..(j-1)]$ contains keys $> A[r]$

- Ideas:
  - $A[j]$ is the current key under examination — $j \geq i$
  - If $A[j] \leq A[r]$, exchange $A[j] \leftrightarrow A[i+1]$ and increment $i$ to maintain the invariant
  - At the end, exchange $A[r] \leftrightarrow A[i+1]$ such that:
    - $A[p..i]$ contains keys $\leq A[i+1]$
    - $A[(i+2)..r]$ contains keys $> A[i+1]$
    - After $A[p..i]$ and $A[(i+2)..r]$ been sorted, $A[p..r]$ is sorted.

- An example: $A[1..8] = \{3, 1, 7, 6, 4, 8, 2, 5\}$, $p = 1$, $r = 8$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 7 | 6 | 4 | 8 | 2 | 5 | $i = 0, j = 1$ |
| 3 | 1 | 7 | 6 | 4 | 8 | 2 | 5 | $i = 1, j = 2$ |
| 3 | 1 | 7 | 6 | 4 | 8 | 2 | 5 | $i = 2, j = 3$ |
| 3 | 1 | 7 | 6 | 4 | 8 | 2 | 5 | $i = 2, j = 4$ |
| 3 | 1 | 4 | 6 | 7 | 8 | 2 | 5 | $i = 3, j = 5$ |
| 3 | 1 | 4 | 6 | 7 | 8 | 2 | 5 | $i = 3, j = 6$ |
| 3 | 1 | 4 | 6 | 7 | 8 | 2 | 5 | $i = 3, j = 7$ |
| 3 | 1 | 4 | 2 | 7 | 8 | 6 | 5 | $i = 4, j = 7$ |
| 3 | 1 | 4 | 2 | 5 | 8 | 6 | 7 | $i = 4, j = 7$ |

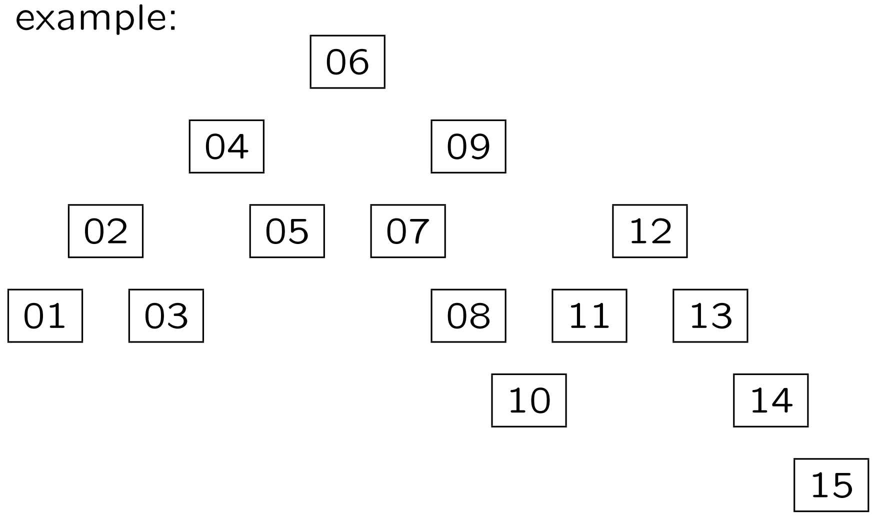
## Quicksort correctness:

- It follows from the correctness of Partition.
- Partition correctness:
  - Loop invariant:
    At the start of `for` loop:
    1. $A[p..i] \leq A[r]$ — $A[s] \leq A[r], p \leq s \leq i$
    2. $A[(i+1)..(j-1)] > A[r]$
    3. $x = A[r]$
  - Proof of LI: (pages 147 – 148)
    1. Initialization
    2. Maintenance
    3. Termination
  - LI correctness implies Partition correctness
- Why we study QuickSort and its analysis:
  - very efficient, in use
  - divide-and-conquer, randomization
  - huge literature
  - a model for analysis of algorithms

## Quicksort recursion tree:

- Observations:
  - (Again) key comparison is the dominant operation
  - Counting KC
    — *only* need to know (at each call) the rank of the split key

- An example:




- More observations:
  - In the resulting recursion tree, at each node
    (all keys in left subtree) $\leq$ (key in this node) $<$ (all keys in right subtree)
  - 1-1 correspondence:
    quicksort recursion tree $\longleftrightarrow$ binary search tree

## Quicksort WC running time:

- The split key is compared with every other key: $(n-1)$ KC

- Recurrence:

$$T(n) = T(n_1) + T(n-1-n_1) + (n-1),$$

  where $0 \leq n_1 \leq n-1$

  Base case: $T(0) = 0$, $T(1) = 0$

- Notice that when both subarrays are non-empty, we will be having

$$(n_1 - 1) + (n - 1 - n_1 - 1) = (n-3)$$

  KC next level ...

- Worst case: one of the subarray is empty !!! needs $(n-2)$ KC next level

- WC recurrence:

$$T(n) = T(0) + T(n-1) + (n-1) = T(n-1) + (n-1),$$

- Solving the recurrence — Master Theorem does NOT apply

$$\begin{aligned} T(n) &= T(n-1) + (n-1) = T(n-2) + (n-2) + (n-1) \\ &= \ldots \\ &= T(1) + 1 + 2 + \ldots + (n-1) \\ &= \frac{(n-1)n}{2} \end{aligned}$$

  So, $T(n) \in \Theta(n^2)$

- Therefore, quicksort is *bad* in terms of WC running time !

## Quicksort BC running time:

- Notice that when both subarrays are non-empty, we will be saving 1 KC ...
- Best case: each partition is a bipartition !!!

  Saving as many KC as possible every level ...

  The recursion tree is as short as possible ...
- Recurrence:

$$T(n) = 2 \times T(\frac{n-1}{2}) + (n-1),$$

- Solving the recurrence — apply Master Theorem? not exactly

  $T(n) \in \Theta(n \log n)$
- Question:
  - What is the best case array? for $n = 7$?
- Conclusion:
  - In order to save time, $A[n]$ better BI-partitions the array ...

    — usually it might not bipartition ... we will push it by a technique called *randomization* (future lectures)

## Quicksort BC running time (cont'd):

- In the recursion tree, what is the number of KC at each level?

  Answer:

  - $n-1$ at the top level
  - at most 2 nodes at the 2nd level, at least $(n_1-1)+(n-1-n_1-1)=n-3$ KC
  - at most 4 nodes at the 3rd level, at least $(n_1-3)+(n-1-n_1-3)=n-7$ KC
  - ...
  - at $k$th level, at most $2^{k-1}$ nodes, at least $n-2^k+1$ KC

- How many levels are there?

  Answer:

  - At least $\lg n$ levels — binary tree

- So, at least we need $\sum_{i=1}^{\lg n-1}(n-2^i+1)$ KC, and

$$\sum_{i=1}^{\lg n-1}(n-2^i+1)=(n+1)(\lg n-1)-(n-2)\in\Theta(n\log n)$$

- Try $n=2^k-1$ to get the closed form for the following recurrence

$$T(n)=\begin{cases}0, & \text{if } n=1\\ (n-1)+T(\lfloor\frac{n-1}{2}\rfloor)+T(\lceil\frac{n-1}{2}\rceil), & \text{if } n\geq 2\end{cases}$$

## Quicksort AC running time:

- Recall the Quicksort algorithm

  Pseudocode:

  **procedure** Quicksort$(A, p, r)$ **p 146

  **if** $p < r$ **then**
  $q \leftarrow$ Partition$(A, p, r)$
  Quicksort$(A, p, q - 1)$
  Quicksort$(A, q + 1, r)$

- The recurrence for running time is:

$$T(n) = \begin{cases} 0, & \text{when } n = 0, 1 \\ T(n_1) + T(n - 1 - n_1) + (n - 1), & \text{when } n \geq 2 \end{cases}$$

- Average case: "What is the probability for the left subarray to have size $n_1$?"

  Average case: always ask "average over what input distribution?"

  - Unless stated otherwise, assume each possible input equiprobable

    *Uniform distribution*

  - Here, each of the _____ possible inputs equiprobable

  - Key observation: equiprobable inputs imply for each key, rank among keys so far is equiprobable

    So, $n_1$ can be $0, 1, 2, \ldots, n-2, n-1$, with the same probability $\frac{1}{n}$

## Solving $T(n)$:

- $$\begin{aligned} T(n) &= \tfrac{1}{n}(T(0)+T(n-1)) \\ &\quad + \tfrac{1}{n}(T(1)+T(n-2)) \\ &\quad + \dots \\ &\quad + \tfrac{1}{n}(T(n-2)+T(1)) \\ &\quad + \tfrac{1}{n}(T(n-1)+T(0)) \\ &\quad + (n-1) \\ &= \tfrac{2}{n}\sum_{i=0}^{n-1} T(i) + (n-1) \end{aligned}$$

- Therefore,
  - $n \times T(n) = 2\sum_{i=0}^{n-1} T(i) + n(n-1)$
  - $(n-1) \times T(n-1) = 2\sum_{i=0}^{n-2} T(i) + (n-1)(n-2)$

- Subtract the two terms:

  $$n \times T(n) - (n-1) \times T(n-1) = 2T(n-1) + 2(n-1)$$

  Rearrange it:

  $$nT(n) = (n+1)T(n-1) + 2(n-1)$$

## Solving $T(n)$ (cont'd):

- Or we can say:

$$\begin{aligned}\tfrac{T(n)}{n+1} &= \tfrac{T(n-1)}{n} + \tfrac{2(n-1)}{n(n+1)} \\ &= \tfrac{T(n-1)}{n} + \tfrac{2}{n+1} - 2(\tfrac{1}{n} - \tfrac{1}{n+1}) \\ &= \tfrac{T(n-1)}{n} + \tfrac{4}{n+1} - \tfrac{2}{n}\end{aligned}$$

  which gives you (iterated substitution)

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

  Recall that $\sum_{i=1}^{n} \frac{1}{i} = H_n = \ln n + \gamma$ — the Harmonic number where $\gamma \approx 0.577\cdots$

- So, from

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

  we have

$$\begin{aligned}T(n) &= 2(n+1)H_{n+1} - (4n+2) \\ &\approx 2(n+1)(\ln(n+1)+\gamma) - (4n+2) \\ &\in \Theta(n \log n)\end{aligned}$$

- Conclusion:

  Quicksort AC running time in $\Theta(n \log n)$.

## Quicksort Improvement and space requirement:

- Quicksort is considered an in-place sorting algorithm:
  - extra space required at each recursive call is only constant.
  - whereas in Mergesort, at each recursive call up to $\Theta(n)$ extra space is required.
- To improve the algorithm, it's better to pick the median as the pivot (but this is difficult)
- Other solution: use a random element as the pivot at every iteration! Pick one $1 \leq i \leq n$ randomly and then exchange $A[i] \leftrightarrow A[n]$ before calling the Partition method.
- This way, no single input is always bad.

## Sorting Algorithms So Far: Running Time Comparison

| Alg. | BC | WC | AC |
|---|---|---|---|
| InsertionSort | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| MergeSort | $\Theta(n \log n)$ | $\Theta(n \log n)$ | ? |
| HeapSort | $\Theta(n \log n)$ | $\Theta(n \log n)$ | ? |
| QuickSort | $\Theta(n \log n)$ | $\Theta(n^2)$ | $\Theta(n \log n)$ |

## Two useful trees in algorithm analysis:

- Recursion tree
  - node $\longleftrightarrow$ recursive call
  - describes algorithm execution for one particular input by showing all calls made
  - one algorithm execution $\longleftrightarrow$ all nodes (a tree)
  - useful in analysis:
    sum the numbers of operations over all nodes

# Recursion tree example:

- Mergesort pseudocode

Merge$(A; lo, mid, hi)$
    **pre-condition: $lo \le mid \le hi$
    **pre-condition: $A[lo, mid]$ and $A[mid+1, hi]$ sorted
    **post-condition: $A[lo, hi]$ sorted

MergeSort$(A; lo, hi)$

  if $lo < hi$ then
    $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
    MergeSort$(A; lo, mid)$
    MergeSort$(A; mid + 1, hi)$
    Merge$(A; lo, mid, hi)$

[1] [4]

[2] [3] [5] [6]

- For different input instance, the number of operations at each node could be different.

## Two useful trees in algorithm analysis:

- Recursion tree
  - node $\longleftrightarrow$ recursion call
  - describes algorithm execution for one particular input by showing all calls made
  - one algorithm execution $\longleftrightarrow$ all nodes (a tree)
  - useful in analysis:
    sum the numbers of operations over all nodes
- Decision tree
  - node $\longleftrightarrow$ algorithm decision
  - describes algorithm execution for all possible inputs by showing all possible algorithm decisions
  - one algorithm execution $\longleftrightarrow$ one root-to-leaf path
  - useful in analysis:
    sum the numbers of operations over nodes on one path

## Selectionsort decision tree:

- Assume input keys in array $A[1..3] = \{a, b, c\}$
- Tree node: if $A[k] > A[j]$ — 2-way key comparison
- Node label $A[j]$

```
SelectionSort(A; n)

if n ≥ 1 then
   for j ← n downto 2 do
      psn ← j
      for k ← j − 1 downto 1 do
         if A[k] > A[psn] then
            psn ← k
      exchange A[j] ↔ A[psn]
   return
```

$A[2] > A[3]$?

No. Yes.

$\boxed{\texttt{abc}}$ $\boxed{\texttt{bac}}$ $\boxed{\phantom{\texttt{abc}}}$ $\boxed{\texttt{bca}}$ $\boxed{\texttt{acb}}$ $\boxed{\texttt{cab}}$ $\boxed{\texttt{cba}}$ $\boxed{\phantom{\texttt{abc}}}$

- In every case — whatever input instance is, 3 KC !!!

## Sorting lower bound:

- Comparison-based sort: keys can be (2-way) compared only !
- This lower bound argument considers only the comparison-based sorting algorithms. For example,
  - Insertionsort, Mergesort, Heapsort, Quicksort,
- Binary tree facts:
  - Suppose there are $t$ leaves and $k$ levels. Then,
  - $t \leq 2^{k-1}$
  - So, $\lg t \leq (k-1)$
  - Equivalently, $k \geq 1 + \lg t$
    — binary tree with $t$ leaves has *at least* $(1 + \lg t)$ levels
- Comparison-based sorting algorithm facts:
  - Look at its *Decision Tree*. We have,
  - It's a binary tree.
  - It should contain every possible permutation of the positions $\{1, 2, \ldots, n\}$.
  - So, it contains at least $n!$ leaves ...
  - Equivalently, it has at least $1 + \lg(n!)$ levels.
  - A longest root-to-leaf path of length at least $\lg(n!)$.
  - The worst case number of KC is at least $\lg(n!)$.
  - $\lg(n!) \in \Theta(n \log n)$
- Therefore, Mergesort and Heapsort are asymptotically optimal (comparison-based) sorting algorithms.

## Algorithm Design Techniques:

- Three major design techniques are:
  1. Greedy method
  2. Divide and Conquer
  3. Dynamic Programming

## 1- Greedy Method:

- This is usually good for optimization problems.

- In an optimization problem we are looking for a solution while we maximize or minimize an objective function. For example, maximizing the profit or minimizing the cost.

- Usually, coming up with a greedy solution is easy; But often it is more difficult to prove that the algorithm is correct.

- General scheme: always make choices that look best at the current step; these choices are final and are not changed later.

- Hope that with every step taken, we are getting closer and closer to an optimal solution.

# Example 1: Fractional Knapsack

- Suppose we have a set $S$ of $n$ items, each with a profit/value $b_i$ and weight $w_i$.
- We also have a knapsack of capacity $W$,
- Assume that each item can be picked at any fraction, that is we can pick $0 \leq x_i \leq w_i$ amount of item $i$.
- Our goal is to fill the knapsack (without exceeding its capacity) with a combination of the items with maximum profit.
- Formally, find $0 \leq x_i \leq w_i$ for $1 \leq i \leq n$ such that $\sum_{i=1}^{n} x_i \leq W$ and $\sum_{i=1}^{n} \frac{x_i}{w_i} \times b_i$ is maximized.
- Greedy idea: start picking the items with more "value":

$$\text{value} \equiv \frac{b_i}{w_i}$$

So let $v_i = \frac{b_i}{w_i}$. The algorithm will be as follows:

- The pseudocode is:

  **Procedure Frac-Knapsack** $(S, W)$

  **for** $i \leftarrow 1$ **to** $n$ **do**
  $\quad x_i \leftarrow 0$
  $\quad v_i \leftarrow \frac{b_i}{w_i}$
  $CurrentW \leftarrow 0$
  **While** $CurrentW < W$ **do**
  $\quad$ let $a_i$ be the next item in $S$ with highest value
  $\quad x_i \leftarrow \min\{w_i, W - CurrentW\}$
  $\quad$ add $x_i$ amount of $i$ to knapsack
  $\quad CurrentW \leftarrow CurrentW + x_i$

- How to find next highest value in each step?

- One way is to sort $S$ at the begining based on $v_i$'s in non-increasing order.

- Another way is to keep a PQ (max-heap) based on values.

- Since we check at most $n$ items, the total time is $O(n \log n)$.

## Correctness of the algorithm:

- We prove, for all $i \geq 0$, that if we have picked $x_1, \ldots, x_i$ from items $1, \ldots, i$ in the first $i$ iterations (respectively), then this partial solution can be extended to an optimal solution.

- In other words, there is some optimal solution call it OPT which has $x_j$ amount from item $j$ for $1 \leq j \leq i$.

## Fractional Knapsack (cont'd)

- We prove by induction on $i$. Base case $i = 0$ we have an empty solution and clearly can be extended to an optimal one.
- Induction Step: Assume the statement is true for $< i$, with $i \geq 1$, prove it for iteration $i$.
- That is, we have picked $x_1, \ldots, x_{i-1}$ of items $1, \ldots, (i-1)$, so does the OPT.
- If OPT picks $x_i$ from item $i$ we are done. So assume OPT picks $x'_i \neq x_i$.
- Note that the algorithm picks maximum amount we can from item $i$ (either $x_i = w_i$ or knapsack is full). Thus $x'_i$ cannot be more than $x_i$, i.e. $x'_i < x_i$.
- Since $W \geq \sum_{k=1}^{i} x_k > \sum_{k=1}^{i-1} x_k + x'_i$, OPT must contain amounts from other items to match the deficiency of $x_i - x'_i$, say amounts $x_{j_1}, x_{j_2}, \ldots, x_{j_\ell}$ of items $j_1, \ldots, j_\ell$ with $x_{j_1} + \ldots + x_{j_\ell} \geq x_i - x'_i$.
- Since items are ranked based on value, all $v_{j_k} \leq v_i$ for $1 \leq k \leq \ell$.
- So if we replace a total of $x_i$ amount from items $j_1, \ldots, j_\ell$ with $x_i$ amount of $i$ in the OPT, the total value does not decrease, $\longrightarrow$ we sill have an optimal solution.
- Now OPT has $x_i$ amount of $i$ and so extends our greedy solution. This completes the induction.