ECM5605(5075) S'20 Algorithms

Lecture 02: *Recurrence Solving, Heap Sort and Quick Sort*

Divide-and-Conquer (revisited)

- The divide-and-conquer paradigm
  - **Divide** the problem into a number of subproblems.
  - **Conquer** the subproblems (solve them).
  - **Combine** the subproblem solutions to derive the solution to the original problem.
- Merge sort: $\mathrm{T}(n) = 2\mathrm{T}(n/2) + \mathrm{O}(n) = \mathrm{O}(n\lg n)$.
  - **Divide** the n-element sequence to be sorted into two $n/2$-element sequence.
  - **Conquer:** sort the subproblems, recursively using merge sort.
  - **Combine:** merge the resulting two sorted $n/2$-element sequences.

Before We Go Further …

- Insertion sort
  - a in-place algorithm: using a small, constant amount of extra storage space.
  - loop-invariant: the partial correctness won't be altered after the loop has run
  - worst-case: $\mathrm{O}(n^2)$, average-case: $\mathrm{O}(n^2)$, best-case: $\mathrm{O}(n)$
- Merge sort:
  - $\in$in-place algorithms?? ⇨ Yes and/or No
  - loop-invariant?? ⇨ Yes
  - worst-case: $\mathrm{O}(n\lg n)$, average-case: $\mathrm{O}(n\lg n)$, best-case: $\mathrm{O}(n\lg n)$

Analyzing Divide-and-Conquer

- Recurrence for a divide-and-conquer algorithms

$$\mathrm{T}(n) = \begin{cases} \Theta(1), & \text{if } n \le c \\ a\mathrm{T}(n/b) + \mathrm{D}(n) + \mathrm{C}(n), & \text{otherwise} \end{cases}$$

  - a: # of subproblems
  - n/b: size of the subproblems
  - $\mathrm{D}(n)$: time to divide the problem of size n into sub-problems
  - $\mathrm{C}(n)$: time to combine the subproblem solutions to derive the answer for the problem of size n

Analyzing Two-way Merge Sort

- Merge sort:

$$\mathrm{T}(n) = \begin{cases} \mathrm{O}(1), & \text{if } n \le c \\ 2\mathrm{T}(n/2) + \mathrm{O}(1) + \mathrm{O}(n), & \text{otherwise} \end{cases}$$

  – $a = 2$: two subproblems
  – $n/b = n/2$: each subproblem has size $\approx n/2$
  – $\mathrm{D}(n) = \Theta(1)$: compute midpoint of array
  – $\mathrm{C}(n) = \Theta(n)$: merging by scanning sorted sub-arrays

Another Example: Binary Search

- Binary search on a sorted array:
  – **Divide:** Check middle element.
  – **Conquer:** Search the subarray.
  – **Combine:** Trivial.
- Recurrence: $\mathrm{T}(n) = \mathrm{T}(n/2) + \mathrm{O}(1) = \mathrm{O}(\lg n)$

$$\mathrm{T}(n) = \begin{cases} \mathrm{O}(1), & \text{if } n \le c \\ \mathrm{T}(n/2) + \mathrm{O}(1) + \mathrm{O}(n), & \text{otherwise} \end{cases}$$

  – $a = 1$: search one sub-array
  – $n/b = n/2$: each subproblem has size $\approx n/2$
  – $\mathrm{D}(n) = \mathrm{O}(1)$: compute midpoint of array
  – $\mathrm{C}(n) = \mathrm{O}(1)$: trivial

Solving Recurrences

- 3 general methods for solving recurrences
  – **Substitution:** Guess a solution and verify it by induction.
  – **Recursion-Tree:** Convert the recurrence into a summation by expanding some terms and then bound the summation
  – **Master Theorem:** if the recurrence has the form
  $\mathrm{T}(n) = a\mathrm{T}(n/b) + f(n)$
  ⇨ most likely there is one formula
- Two simplifications that won't affect
  – ignore floors and ceilings
  – assume base cases are constant, i.e., $\mathrm{T}(n) = \mathrm{O}(1)$ for small n

Method 1: Substitution

- The most general methods:
  1. **Guess** the form of the solution
  2. **Verify** by induction
  3. **Solve** the constants
- Example: $\mathrm{T}(n)=4\mathrm{T}(n/2)+n$
  – Assume $\mathrm{T}(1) = \mathrm{O}(1)$
  – Guess $\mathrm{O}(n^3)$ [Prove O and Ω separately]
  – Assume $\mathrm{T}(k) \le ck^3$ for $k < n$
  – Prove $\mathrm{T}(n) \le cn^3$ by induction

Example of Substitution

- Basis
  $T(2) = 4T(1)+2 = 6 \le c*2^3$ (pick $c=1$)
- Assume $T(k) \le ck^3$ for $k < n$
- Prove $T(n) \le cn^3$ by induction
  $T(n) = 4T(n/2) + n$
  $\le 4c(n/2)^3 + n$
  $= cn^3/2 + n$
  $= cn^3 - (cn^3/2-n)$ // desired-residual
  $\le cn^3$ // desired □
  where the residual $cn^3/2 - n \ge 0$
- For example, if $c \ge 2$ and $n \ge 1$

A Tighter Upper Bound

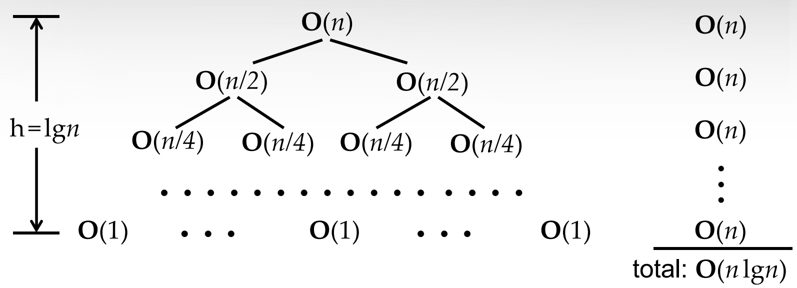
- Guess $T(n) = 4T(n/2)+n = O(n^2)$
- Assume $T(k) \le ck^2$ for $k < n$
  $T(n) = 4T(n/2) + n$
  $\le 4c(n/2)^2 + n$
  $= cn^2 + n = O(n^2)$ // true but not proved
  $= cn^2 - (-n)$ // no $c>0$ ⇨ residual>0
  $\not\le cn^2$
  ⇨ Wrong!! We cannot complete the induction.
- Solution: strengthen the inductive hypothesis
  ⇨ *subtract* a low-order term

Strengthened Inductive Hypothesis

- Guess $T(n) = 4T(n/2)+n = O(n^2) = c_1 \times n^2 - c_2 \times n$
- Basis
  – $T(2) = 4T(1)+2 = 6 \le c_1 \times 2^2 - c_2 \times 2$
  – pick $c_1=4$ and $c_2=3$
- Assume $T(k) \le c_1k^2 - c_2k$ for $k < n$
  $T(n) = 4T(n/2) + n$
  $\le 4(c_1(n/2)^2 - c_2(n/2)) + n$
  $= c_1n^2 - 2c_2n + n$
  $= c_1n^2 - c_2n - (c_2n - n)$ // pick $c_2 > 1$
  $\le c_1n^2 - c_2n$ □

Method 2: Recursion Tree

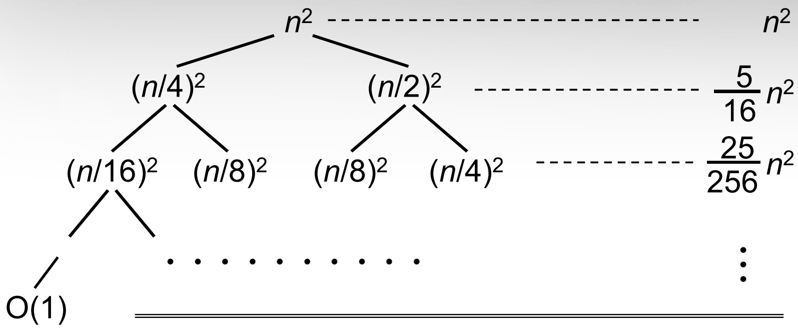
- Visualize the costs (time) of recursive execution in the algorithm



- Can be unreliable because using ellipses(…)
- Follows human being's intuition and good for generating guess for the substitution method.

An Example of Recursion Trees

- Solve $\mathrm{T}(n) = \mathrm{T}(n/4) + \mathrm{T}(n/2) + n^2$



$$\begin{aligned}\mathrm{T}(n) &= n^2(1 + (\tfrac{5}{16})^1 + (\tfrac{5}{16})^2 + (\tfrac{5}{16})^3 + \cdots) \\ &= \mathrm{O}(n^2)\end{aligned}$$

Iterative Expression

- Example $\mathrm{T}(n) = 4\mathrm{T}(n/2)+n$

$$\begin{aligned}\mathrm{T}(n) &= 4\mathrm{T}(n/2) + n \\ &= 4(4\mathrm{T}(n/4) + n/2) + n \\ &= 16\mathrm{T}(n/4) + 2n + n \\ &= 64\mathrm{T}(n/8) + 4n + 2n + n \\ &= 4^{\lg n}\mathrm{T}(1) + 2^{\lg n - 1}n + \ldots + 4n + 2n + n \\ &= 4^{\lg n}c + n\sum_{k=0}^{\lg n - 1}2^k \\ &= cn^{\lg 4} + n(2^{\lg n} - 1) \\ &= cn^2 + n(n^{\lg 2} - 1) \\ &= (c+1)n^2 - n \\ &= \Theta(n^2)\end{aligned}$$

// remember $a^{\lg \mathbf{b}} = b^{\lg \mathbf{a}}$

□

Method 3: Master Theorem

- The master theorem applies to the recurrences of the form:

  $\mathrm{T}(n) = a\mathrm{T}(n/b) + f(n)$

  where

  $a \geq 1$, $b > 1$ and

  $f(n)$ is asymptotically positive

- Bound $\mathrm{T}(n)$ by comparing $f(n)$ with $n^{\log_b a}$
  – three cases: >, = and <

Compare $f(n)$ with $\mathrm{O}(n^{\log_b a})$

- (Case 1)

  If $f(n) = \mathrm{O}(n^{\log_b a - \epsilon})$ for some constant $\varepsilon > 0$

  – $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by a n^{ε} factor)

  then the solution: $\mathrm{T}(n) = \mathrm{O}(n^{\log_b a})$

- (Case 2)

  If $f(n) = \mathrm{O}(n^{\log_b a} \times (\lg n)^k)$ for some constant $k \geq 0$

  – $f(n)$ and $n^{\log_b a}$ grow at the same rate.

  then the solution: $\mathrm{T}(n) = \mathrm{O}(n^{\log_b a} \times (\lg n)^{k+1})$

Compare $f(n)$ with $O(n^{\log_b a})$ (cont'd)

- (Case 3)

  If $f(n) = O(n^{\log_b a+\epsilon})$ for some constant ε>0

  – $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by a n^{ϵ} factor)

  and $f(n)$ satisfies the regularity condition:

  $af(n/b) \le cf(n)$ for some constant $c<1$

  – then the solution: $T(n) = O(f(n))$

Proof of Master Theorem (1/3)

$T(n) = aT\left({}^{n}/_{b}\right) + cn^k, a, b, c, k \in \mathbb{R}$

$Let\ a = b^m,$

$$T(n) = aT\left({}^{n}/_{b}\right) + cn^k$$

$$= a\left(aT\left({}^{n}/_{b^2}\right) + c\left({}^{n}/_{b}\right)^k\right) + cn^k$$

$$= a\left(a\left(aT\left({}^{n}/_{b^3}\right) + c\left({}^{n}/_{b^2}\right)^k\right) + c\left({}^{n}/_{b}\right)^k\right) + cn^k$$

$$= a\left(a\left(\cdots T\left({}^{n}/_{b^m}\right) + c\left({}^{n}/_{b^{m-1}}\right)^k\right) + \cdots\right) + cn^k,$$

$when\ {}^{n}/_{b^m} = 1 \Rightarrow n = b^m$

Proof of Master Theorem (1/3)

$Assume\ T(1) = C,$

$$T(n) = ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + \cdots + cb^{mk}$$

$$= c\sum_{i=0}^{m} a^{m-i}b^{ik} = ca^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i$$

$(case1)\ if\ a > b^k\ \left(\frac{b^k}{a} < 1\right),$

$T(n) = O(a^m)$

$\because m = \log_b n, a^m = a^{\log_b n} = n^{\log_b a}$

$\therefore T(n) = O(a^m) = O(n^{\log_b a})$

Proof of Master Theorem (3/3)

$(case2)\ if\ a = b^k\ \left(\frac{b^k}{a} = 1\right),$

$\because k = \log_b a, m = \log_b n$

$\therefore T(n) = O(a^m \cdot m)$
$= O\left(a^{\log_b n} \cdot \log_b n\right)$

$= O\left(n^{\log_b a} \cdot \log_b n\right)$
$= O(n^k \cdot \log n)$

$(case3)\ if\ a < b^k\ \left(\frac{b^k}{a} > 1\right),$

$let\ F = \left(\frac{b^k}{a}\right),$

$m = \log_b n$

$T(n)$

$= O\left(a^m \cdot \frac{F^{m+1} - 1}{F - 1}\right)$

$= O(a^m \cdot F^m) = O\left(b^{km}\right)$

$= O\left((b^m)^k\right) = O(n^k)$

Examples of Using Master Theorem

- (Ex 1) $T(n)=4T(n/2)+n$
  - $a=4$, $b=2$ ⇨ $n^{\log_b a}=n^2$ and $f(n)=n$
  - apply to case 1: $f(n)=O(n^{2-\varepsilon})$ for $\varepsilon=1$
  - $\therefore T(n) = O(n^2)$
- (Ex 2) $T(n)=4T(n/2)+n^2$
  - $a=4$, $b=2$ ⇨ $n^{\log_b a}=n^2$ and $f(n)=n^2$
  - apply to case 2: $f(n)= O(n^2\lg^0 n)$ for $k=0$
  - $\therefore T(n) = O(n^2\lg n)$

Examples of Using Master Theorem (cont'd)

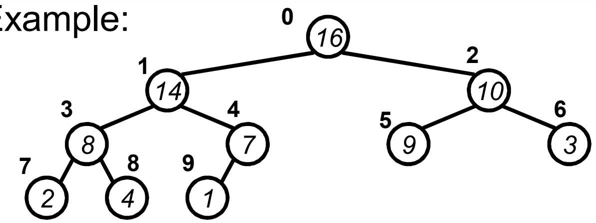
- (Ex 3) $T(n)=3T(n/4)+ n\lg n$
  - $a=3$, $b=4$ ⇨ $n^{\log_b a}=n^{0.79}$ and $f(n)= n\lg n$
  - apply to case 3:
    $f(n)= O(n^{0.79+\varepsilon})$ for some ε
  - Check regularity condition
    $3(n/4)\times\lg(n/4) \le cn\lg n$ for $c=3/4$
  - $\therefore T(n) = O(n\lg n)$
- (Ex 4) $T(n)=4T(n/2)+ n^2/\lg n$
  - Master theorem does not apply but recursion tree does ⇨ $T(n)= O(n^2\lg\lg n)$

Summary (Part 1)

- Three recurrence solving methods
  1. Substitution
  2. Recursion Tree
  3. Master Theorem
- Little quiz: $T(n) = 2T(\sqrt{n}) + \lg n$
  - Hint: let $m=\lg n$
  - Answer: $T(n) = O(\lg n\lg\lg n)$
- Up next
  - Heap sort and quick sort

Binary Heaps

- A binary *heap* is a "complete" binary tree, usually represented as an ***array***:
  - Parent(i) { return $\lceil {}^{i}/_{2} \rceil$-1; }
  - LeftChild(i) { return 2*i+1; }
  - RightChild(i) { return 2*i+2; }
- Example:



A =

16	14	10	8	7	9	3	2	4	1

Max-Heap Property

- Heaps need to satisfy the max-*heap property*:
  A[*Parent*(*i*)] ≥ A[*i*] for all nodes *i* > 1
  - In other words, the value of a node is at most the value of its parent
  - The largest value is thus stored at the root (A[0])
- Because the heap is a binary tree, the height of any node is at most ($\lg n$)

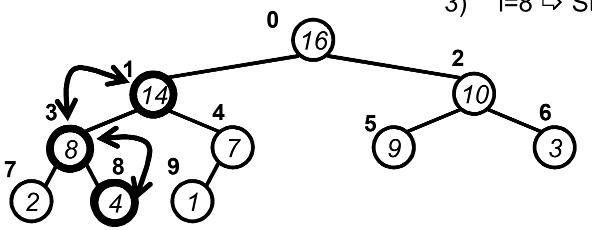
Max-Heapify() Idea

- **Max-Heapify()**: maintain max-heap property
  - Given: a node i in the heap with children l and r
  - Given: two subtrees rooted at l and r, assumed to be heaps
  - Action: let the value of the parent node “float down” so subtree at i satisfies the heap property
    - If $A[i] < A[l]$ or $A[i] < A[r]$, swap $A[i]$ with the largest of $A[l]$ and $A[r]$
    - Recursive walk on that subtree
  - Running time: $O(h)$, h = height of heap = $O(\lg n)$

Max-Heapify() Example

```
Max-Heapify(A, i)
    l ← LEFT(i)
    r ← RIGHT(i)
    if l ≤ heap-size[A] and A[l] > A[i]
        then largest ← l
        else largest ← i
    if r ≤ heap-size[A] and A[r] >
A[largest]
        then largest ← r
    if largest ≠ i
        then exchange A[i] ↔ A[largest]
                Max-Heapify(A, largest)
```

1) i=1 ⇨ l=3 & r= 4 ⇨largest=3 ⇨swap A[1] and A[3] & call Max-Heapify(A,3)
2) i=3 ⇨ l=7 & r= 8 ⇨largest=8 ⇨swap A[3] and A[8] & call Max-Heapify(A,8)
3) i=8 ⇨ Stop!

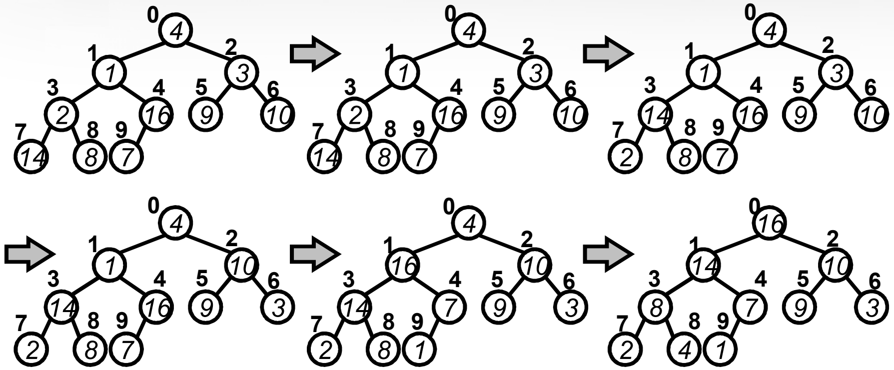


Build-Max-Heap() Idea

- Use **Max-Heapify()** in a bottom-up manner to convert A into a heap
  - Fact: for array of length n, all elements in range $A[\lfloor n/2 \rfloor \ldots n]$ are heaps (***Why?***)
  - Therefore,
    ⇨ Walk backwards through the array from $n/2$-1 to 0, calling **Max-Heapify()** on each node.
    ⇨ Order of processing guarantees that the children of node i are heaps when i is processed

Build-Max-Heap() Example

```
Build-Max-Heap(A)
heap-size[A] ← length[A]
for i ←⌊length[A]/2⌋ downto 1
    do Max-Heapify(A,i)
```



Analyzing Build-Max-Heap()

- Each call to **Max-Heapify()** takes $O(\lg n)$ time
- There are $O(n)$ such calls (precisely, $\lfloor n/2 \rfloor$)
- Thus the running time is $O(n \lg n)$
  - *Is this a correct asymptotic upper bound?*
  - *Is this an asymptotically tight bound?*
- A tighter bound is $O(n)$
  - *How can this be? Is there a flaw in the above reasoning?*

Analyzing Build-Max-Heap(): Tight

- To **Max-Heapify()** a subtree takes $O(h)$ time where h is the height of the subtree
  - $h = O(\lg m)$, m = # nodes in subtree
  - The height of most subtrees is small
- Fact: an n-element heap has at most $\lceil \boldsymbol{n}/2^{\boldsymbol{h}+1} \rceil$ nodes of height h ⇨ *why??*
  - n=10, h=2 ⇨ at most $\lceil n/2^{h+1} \rceil = 2$
- The textbook uses this fact to prove that **Build-Max-Heap()** takes $O(n)$ time
  -

Tight Bound on Build-Max-Heap()

- Ex: For a complete binary tree with 15 nodes
  - Level 0 ≡ height 3 ⇨ node# = 1 = $\lceil 15/2^{3+1} \rceil$
  - Level 1 ≡ height 2 ⇨ node# = 2 = $\lceil 15/2^{2+1} \rceil$
  - Level 2 ≡ height 1 ⇨ node# = 4 = $\lceil 15/2^{1+1} \rceil$
  - Level 3 ≡ height 0 ⇨ node# = 8 = $\lceil 15/2^{0+1} \rceil$

By induction height *h* includes nodes.

$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil O(h)$$

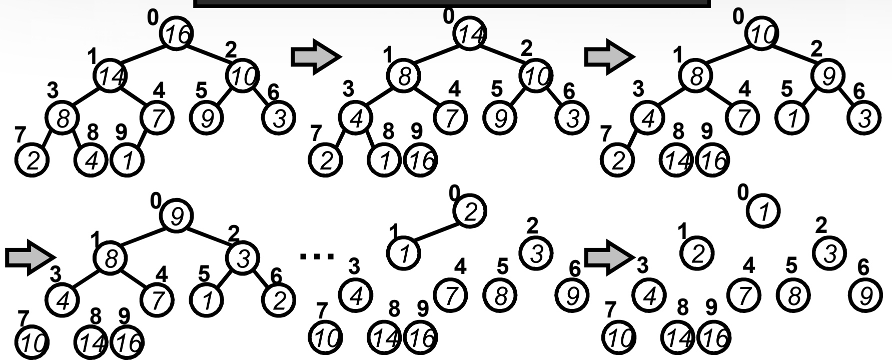
$$= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil h/2^h \right\rceil\right) \left(\because \sum_{k=0}^{\infty} kx^k = x/(1-x)^2, \forall |x| < 1\right) = O\left(\frac{1/2}{(1-1/2)^2}\right) = O(2n) = O(n)$$

Heapsort

- Given **Build-Max-Heap()**, an in-place sorting algorithm is easily constructed:
  - –Maximum element is at $\mathrm{A}[1]$: $\mathrm{O}(1)$
  - –Discard by swapping with element at $\mathrm{A}[n]$
    - –Decrement heap_size[A]:
  - –$\mathrm{A}[n]$ now contains correct value
  - –Restore heap property at $\mathrm{A}[1]$ by calling **Max-Heapify()**
  - –Repeat, always swapping $\mathrm{A}[1]$ for A[heap_size(A)]

Heapsort Example

```
Heapsort(A)
Build-Max-Heap(A)
for i ← length[A]-1 downto 1
    do exchange A[1] ↔ A[i]
    heap-size[A] ← heap-size[A] - 1
    Max-Heapify(A,i)
```



Analyzing Heapsort

- The call to **Build-Max-Heap()** takes $\mathrm{O}(n)$ time
- Each of the n - 1 calls to **Max-Heapify()** takes $\mathrm{O}(\lg n)$ time
- Thus the total time taken by **HeapSort()**

$$\begin{aligned} \mathrm{T(n)} &= \mathrm{O}(n) + (n - 1)\,\mathrm{O}(\lg n) \\ &= \mathrm{O}(n) + \mathrm{O}(n \lg n) \\ &= \mathrm{O}(n \lg n) \end{aligned}$$

Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort (coming next) usually wins
- But the heap data structure is incredibly useful for implementing *priority queues*
  - –A data structure for maintaining a set *S* of elements, each with an associated value or *key*
- ***What might a priority queue be useful for?***
  - –(Maximum-)priority queues: the highest priority element/job/packet should be scheduled first

Priority Queue Operations

- Supports the operations **Insert()**, **Maximum()**, and **ExtractMax()**
- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key
- *How could we implement these operations using a heap?*

Quicksort

- Proposed by C.A.R. Hoare in 1962
- (SIAM news) One of the Best of 20th century: Editors Name Top 10 **Algorithms**
- A ***in-place*** algorithm
- Sorts $O(n \lg n)$ in the average case
- Sorts $O(n^2)$ in the worst case
- *So why would people use it instead of merge sort?*

Quicksort

- A divide-and-conquer algorithm
  - **Divide**: partition array $A[p..r]$ into two non-empty subarrays $A[p..q]$ and $A[q+1..r]$
    - ⇨ Loop invariant: All elements in $A[p..q] \leq$ all elements in $A[q+1..r]$
  - **Conquer**: the subarrays are recursively sorted by calls to **Quicksort**
  - **Combine**: Unlike merge sort, no combining step: two subarrays form an already-sorted array
    - ⇨ An in-place algorithm

Quicksort Pseudocode

Quicksort(A, p, r)

```
{
    if (p < r)
    {   // divide A into two subproblems
        q ← Partition(A, p, r);  // q: key to partition
        Quicksort(A, p, q-1);  // solve 1st subproblem
        Quicksort(A, q+1, r);  // solve 2nd subproblem
    }
}
```

Initial Call: **Quicksort**(A,1,length[A]+1)

Partition()

- Clearly, all the action takes place in the **Partition**() function
  - Rearranges the subarray in place
  - End result:
    - ⇨ Two subarrays

    - ⇨ All values in first subarray ≤ all values in second
  - Returns the index of the "pivot" element separating the two subarrays
- Q: How do you implement this function?

Partition() in Words

- **Partition**(A, l, r):
  - Select an element to act as the "pivot (A[k])" (*which one?*)
  - Grow two subarrays, A[$l..i$] and A[$j..r$]
    - All elements in A[$l..i$] ≤ pivot
    - All elements in A[$j..r$] ≥ pivot
  - Increment index i until A[i] ≥ pivot
  - Decrement index j until A[j] ≤ pivot
  - Swap A[i] and A[j]
  - Repeat until $i \geq j$
  - Return j

Pseudocode (1) of Partition()

```
Partition(A, l, r)
  x ← A[k]  //k: pivot idx
  i ← l and j ← r
  repeat
      repeat
          i ← i+1
      until (A[i] ≥ x)
      repeat
          j ← j–1
      until (A[j] ≤ x)
      if (i < j)
          swap(A[i], A[j])
  until (i ≥ j)
  A[l] ← A[j]
  A[j] ← x
  return j;
```

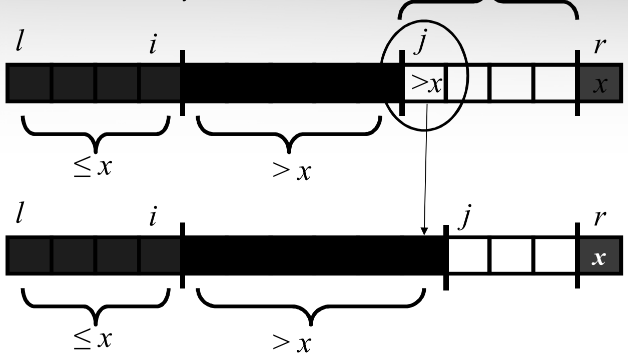

Pseudocode (2) of Partition()

```
Partition(A, l, r)    // k = Partition(A, l, r)
  x ← A[r]            // x: the rightmost as pivot
  i ← l-1
  for j ← l to r-1 do
      if A[j] ≤ x then
          i ← i+1
          swap(A[i], A[j])
  swap(A[i+1], A[r])
  return i+1
```

What is the running time of **Partition()**?
Answer: O(n)

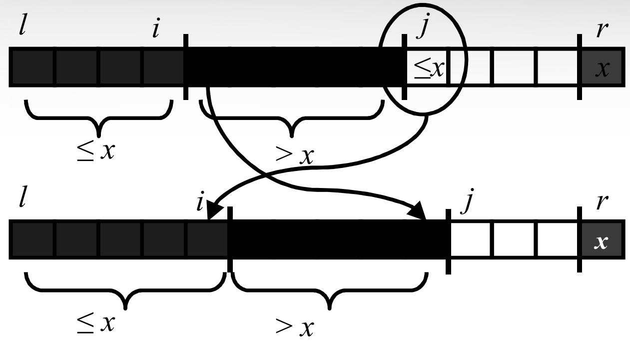
Exchange *i* and *j* in Quicksort

- Case 1: if $A[j] > x = A[r]$

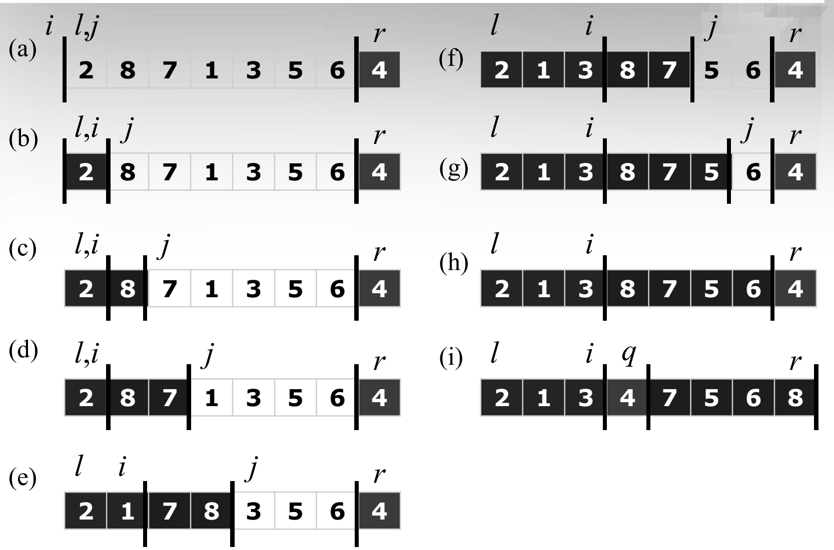


Exchange i and j in Quicksort

- Case 2: if $A[j] \le x = A[r]$



Example of Partition (x=A[r]=4)



Analyzing Quicksort

- *What will be the worst case for the algorithm?*
  Partition is always unbalanced
- *What will be the best case for the algorithm?*
  Partition is perfectly balance
- *Which is more likely?*
  The latter, by far, except...
- *Will any particular input elicit the worst case?*
  Yes: already-sorted input

Quicksort Runtime Analysis

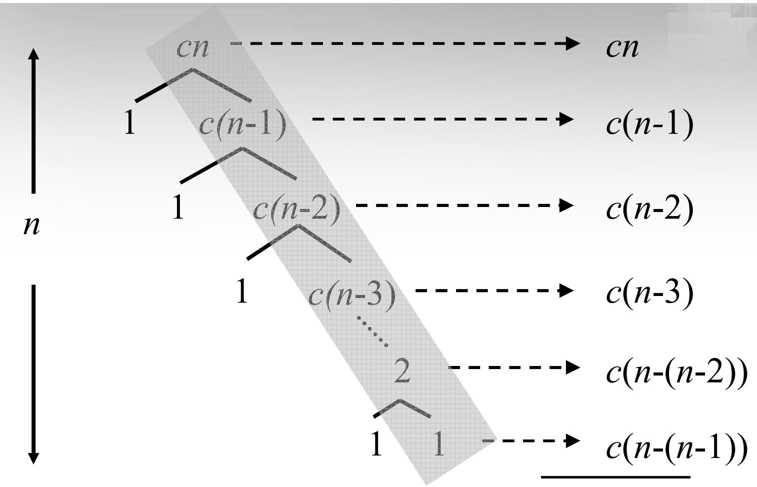
- A divide-and-conquer algorithm:
  - $\mathbf{T}(n)=\mathbf{T}(q-p+1)+\mathbf{T}(r-q)+\mathrm{O}(n)$
- Two key factors to decide its performance
  1. the pivot x picked *during* executing $\mathrm{Partition}(A,p,r)$
  2. the position of q obtained *after* $\mathrm{Partition}(A,p,r)$
- Best-case: perfectly balanced splits
  - each partition results in $(n/2{:}n/2)$ split
  - $\mathbf{T}(n) = \mathbf{T}(n/2)+\mathbf{T}(n/2)+\mathrm{O}(n) = 2\mathbf{T}(n/2)+\mathrm{O}(n)$
  - ⇨ Look familiar?? Look like Mergesort
  - $\therefore \mathbf{T}(n) = \mathrm{O}(n\lg n)$ can proved by …

Worst-case Analysis of Quicksort

- Worst-case: each partition results in $1{:}(n\text{-}1)$ split in a row

$$\begin{aligned}
\mathrm{T}(n) &= \mathrm{T}(1)+\mathrm{T}(n-1)+\mathrm{O}(n)\\
&= \mathrm{T}(1)+(\mathrm{T}(1)+\mathrm{T}(n-2)+\mathrm{O}(n-1))+\mathrm{O}(n)\\
&= \cdots\\
&= n\mathrm{T}(1)+\mathrm{O}(\textstyle\sum_{k=1}^{n} k)\\
&= \mathrm{O}(n^2)
\end{aligned}$$

Visualization of Worst-Case Analysis



$\sum_{k=1}^{n} ck = \mathrm{O}(n^2)$

Formal Worst-case Analysis

- **The *real* upperbound: $\mathrm{O}(n^2)$**

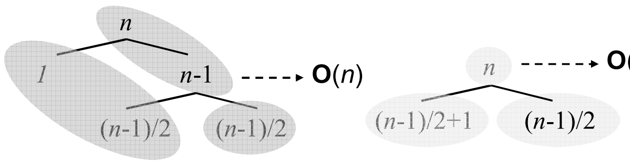
$$\begin{aligned}
\mathrm{T}(n) &= \max_{1\le q\le n} \mathrm{T}(q-1)+\mathrm{T}(n-q)+\mathrm{O}(n)\\
&= \max_{1\le k\le n-1} \mathrm{T}(k)+\mathrm{T}(n-k-1)+\mathrm{O}(n)
\end{aligned}$$

- Guess: $\mathrm{T}(n) \le cn^2 = \mathrm{O}(n^2)$
- By substituting:

$$\begin{aligned}
\mathrm{T}(n) &\le \max_{1\le k\le n-1} \{ck^2+c(n-k-1)^2\}+\mathrm{O}(n)\\
&\le c\times \max_{0\le k\le n-1} k^2+(n-k-1)^2+\mathrm{O}(n)\\
&\le c(n-1)^2+\mathrm{O}(n)\\
&\le cn^2-c(2n-1)+\mathrm{O}(n)\\
&\le cn^2=\mathrm{O}(n^2)
\end{aligned}$$

Average-case Analysis of Quicksort

- Intuition: some splits will be close to balanced and others imbalanced
  - Good and bad splits are randomly distributed in the recursion tree
- Observation: asymptotically bad run time happens only when many bad splits happen in a row
  - A bad split followed by a good split results in a good partition after one additional step
  - $\therefore$ still get $\mathbf{T}(n) = \mathbf{O}(n\lg n)$ with slightly large ***const***.



Randomized Quicksort

- Expect to get average-case behavior of quicksort on all inputs
  - Randomization!!
- Two approaches
  1. Randomly permute input
  2. Choose the pivot randomly at each iteration
     - EX:

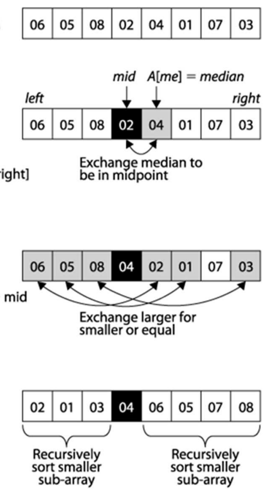
```
RANDOMIZED-PARTITION(A, l, r)
i ← RANDOM(l, r)
exchange A[r] ↔ A[i]
return PARTITION(A, l, r)
```

```
RANDOMIZED-QUICKSORT(A, l, r)
if l < r then
  k ← RANDOMIZED-PARTITION(A, l, r)
  RANDOMIZED-QUICKSORT(A, l, k-1)
  RANDOMIZED-QUICKSORT(A, k+1, r)
```


Exercise – Median Sort

```
sort (A)
  medianSort (A, 0, n − 1)
end

medianSort (A, left, right)
  if (left < right) then
    find median value A[me] in A[left, right]
    mid = ⌊(right+left)/2⌋
    swap A[mid] and A[me]
    for left=0 to mid − 1 do
      if (A[i] > A[mid]) then
        find A[k] ≤ A[mid] where k > mid
        swap A[i] and A[k]
    medianSort (A, left, mid − 1)
    medianSort (A, mid + 1, right)
end
```



Complexity:

worst-case?

Summary (Part 2)

- Heaps and Priority Queues
  - Definition of Max-Heap Property
  - Max-Heapify() and Build-Max-Heapify()
  - Heapsort algorithm and its complexity
- Quicksort
  - The important properties of Quicksort
  - Forget the pseudocode, remember the idea
  - Two key factors to decide its performance
  - Best-case? Worst-case? Average-case?
- Next lecture ⇨ Linear sorts and order statistics