Part 5: Divide and Conquer

Contents

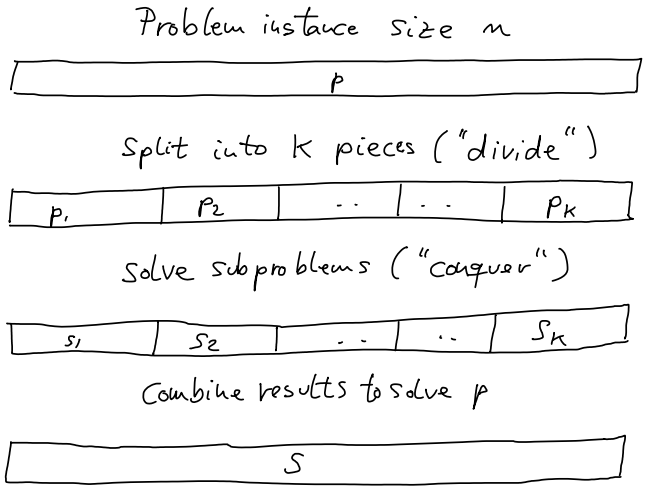


[document finalized]



## Divide and Conquer

The "Divide and Conquer" (D&C) design strategy for algorithms is based on the following general plan:






1. ["Divide" Step ] The problem instance is divided into several smaller instances of the same problem, ideally of about the same size. To obtain fast programs it is important that the size of the biggest remaining subproblem is only a fraction of the original size. In selection sort for instance, the subproblem was only one element smaller than the previous one, leading to runtime $\Theta(n^2)$ — which is SLOW.
2. ["Conquer" Step ] The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough)
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

In what follows we will see a series of classic D&C examples.



## Binary Search

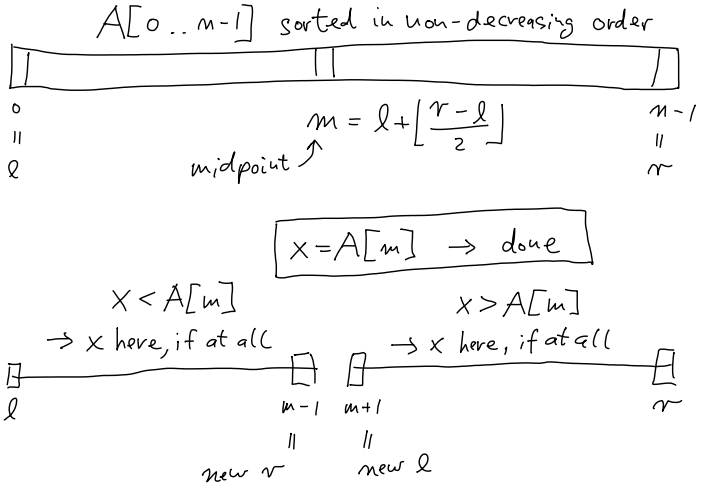
Task: find index of $x$ in array $A[0..n-1]$ if $x$ is stored in $A$, and $-1$ otherwise.

If nothing is known about $A$, then the worst case runtime is $\Theta(n)$, because we have to compare $x$ with every key.

However, if $A$ is sorted we can find values much faster – in time $\Theta(\log n)$.

Example: Telephone books. We want to find a phone number for a given name $x$. Starting in the middle, at any given time in the search we know in what remaining part to continue (left or right) by comparing name $x$ with the first entry on the page – say. We then pick the half-way page in the remaining part and repeat the process, until we narrowed the search down to one page. There we can use a similar strategy: starting in the middle ...

Illustration:






Pseudo code:

```
// assumes n > 0
// A[0..] sorted in non-decreasing order
// returns index of x in A[l..r] if it occurs
// and -1 otherwise
function BinarySearch(A[0..], l, r, x)
if l > r then  // no key
  return -1    // => x not present
end
m <- l + floor((r-l)/2) // middle (*)
if x < A[m] then
  return BinarySearch(A, l, m-1, x)
else if x > A[m] then
  return BinarySearch(A, m+1, r, x)
else
  return m     // found
end
```

Claim: The algorithm is correct, i.e.

1. it terminates for all inputs (runtime is $O(\log n)$)
2. it computes the correct value: `m` $\geq 0$ for an `m` with `A[m] = x`, and $-1$ if $x$ does not occur in `A[l..r]`



Proof:

1: Let $T(n)$ be number of times `BinarySearch` is called for $n = \mathtt{r} - \mathtt{l} + 1 \geq 1$ in the worst case. Then $T(n) \leq \log(n) + 2$ for all $n \geq 1$ and $T(0) = 1$.

Proof by induction on $n$:

Induction base $n = 1$: Inspecting the code yields $T(1) = 2 \leq \log(1) + 2 = 2$. OK.

Induction step: suppose $n \geq 2$ and $T(k) \leq \log(k) + 2$ for all $1 \leq k < n$. The length of the subarray processed by the subsequent `BinarySearch` call depends on whether $n$ is even or odd:

If $n$ is even, the parts have sizes $\frac{n}{2} - 1$ and $\frac{n}{2}$. Hence:

$$T(n) \leq max(T(\frac{n}{2} - 1), T(\frac{n}{2})) + 1$$

Using the induction hypothesis and assuming $n \geq 4$:

$$T(n) \leq max(\log(\frac{n}{2} - 1) + 2, \log(\frac{n}{2}) + 2) + 1$

($\log$ monotone)

$$\leq \log(\frac{n}{2}) + 2 + 1 = \log(n) + 2$$

For $n = 2$ the inequality also holds (using $T(0) = 1$).



If $n$ is odd, both parts have size $\frac{n-1}{2}$ and thus

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

(induction hypothesis)

$$\leq \log(\frac{n-1}{2}) + 2 + 1$$

($\log$ monotone)

$$\leq \log(\frac{n}{2}) + 2 + 1 \ = \ \log(n) + 2$$

Therefore, for all $n \geq 1 : \ T(n) \leq \log(n) + 2$ (and even $T(n) \leq \lfloor \log(n) \rfloor + 2$ because $T(n)$ is an integer). This shows that `BinarySearch` terminates for all $n$.

2: Exercise □

So, `BinarySearch`'s worst case runtime is $O(\log n)$. Similarly, one can show that it also is $\Omega(\log n)$.

The approach we used is called **"decrease and conquer"**, because we reduce the problem size first and then proceed with the smaller problem instance.

## Binary Exponentiation

Lecture 13

Evaluating monomials such as $x^n$ quickly for large $n$ is essential for modern cryptography.

How can we do this fast?

Naive approach: $x^n = \underbrace{x \cdot x \cdots x}_{n \text{ times}}$

Number of multiplications: $n - 1$

This is definitely too slow for $n = 2^{100}$.

But consider this: $x^{16} = (x^8)^2 = ((x^4)^2)^2 = (((x^2)^2)^2)^2$

Each square operation takes just one multiplication. So, $x^{16}$ can be evaluated with $4$ multiplications, rather than $15$ and $x^{1024}$ only requires $10$ multiplications, rather than $1023$ !

Can this idea be generalized to values $n$ that are not powers of $2$ ?



Certainly. We note

$x^0 = 1$ for all $x$, and

$x^n = (x^{\frac{n}{2}})^2$, if $n$ is even, and

$x^n = (x^{\frac{n-1}{2}})^2 \cdot x$, if $n$ is odd.

This looks again like a "decrease and conquer" setup, in which we solve a smaller subproblem and then construct the full solution from that.

```
// computes x raised to the n-th power
// assumes n >= 0
function Power(x, n)
if n = 0 then
  return 1
else if n = 1 then
  return x
end
if n even then
  p <- Power(x, n/2)
  return p*p
else
  p <- Power(x, (n-1)/2)
  return p*p*x
end
```



What is the runtime of `Power(x, n)`?

Claim: Let $T(n)$ the number of times `Power` is executed when invoked with `Power(x, n)`.
Then $T(n) \leq \log(n) + 1$ for all $n \geq 1$.

Proof: Induction on $n$.

Induction Base $n = 1$:

Inspect code: $T(1) = 1 \leq \log(1) + 1 = 1$. OK.

Induction Step:

Suppose $n \geq 2$ and $T(k) \leq \log(k) + 1$ for all $k < n$.

In case $n$ is even:

$T(n) = T(\frac{n}{2}) + 1 \leq (\log(\frac{n}{2}) + 1) + 1 = \log(n) + 1$

In case $n$ is odd:

$T(n) = T(\frac{n-1}{2}) + 1 \leq (\log(\frac{n-1}{2}) + 1) + 1 \leq \log(n) + 1$

Therefore, the claim is true. □



The worst case runtime of `Power(x,n)` is therefore $O(\log n)$. With a similar argument one can also show that it is $\Omega(\log n)$. Thus, the worst case runtime is $\Theta(\log n)$. This allows us to evaluate $x^n$ quickly even for large values of $n$.

At the core of the `Power` function is a test that checks whether $n$ is even or odd. Modern computers represent numbers in base 2 using digits 0 and 1. In this representation, the lowest order bit tells us whether the number is odd (1) or even (0). Also, division by 2 followed by rounding down is very fast: it's just a right shift of all digits (similar to dividing by 10 and rounding down when using base 10, e.g. $\lfloor 1234/10 \rfloor = 123$)

`Power(x,n)` is not optimal in terms of executed multiplications. If reusing prior results is allowed, the smallest example is $n = 15$:

`Power(x,15)`: $x^{15} = x(x(x \cdot x^2)^2)^2$ : 6 multiplications

but also $x^{15} = x \cdot (x^2) \cdot ([x^3]^2)^2$ : 5 multiplications

## Fast Matrix Multiplication

Matrix multiplication is a fundamental operation

All standard linear algebra operations such as

- Inverting Matrices
- Computing Determinants
- Solving Systems of Linear Equations

are of similar complexity.

How fast can we multiply matrices?

We consider matrix multiplications over rings:



## Definition: (Ring)

A ring is an algebraic structure $(S, +, \cdot, 0, 1)$ in which $S$ is a set of elements with $0, 1 \in S$ and $+$ and $\cdot$ are binary operations on $S$ with the following properties:

- $+$ and $\cdot$ are associative
- $+$ is commutative
- $\cdot$ distributes over $+$
  $(a+b)\cdot c = (a\cdot c)+(b\cdot c)$ and $a\cdot(b+c) = (a\cdot b)+(a\cdot c)$
- 0 is the neutral element for $+$ $(a + 0 = 0 + a = a)$
- 1 is the neutral element for $\cdot$ $(a \cdot 1 = 1 \cdot a = a)$
- For each $a \in S$ there is an inverse $-a \in S$ such that $a + (-a) = (-a) + a = 0$

Examples: $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$ are rings, $\mathbb{N}$ is not.



Let $\mathcal{R} = (R, +, \cdot, 0, 1)$ be a ring and $M_n$ the set of $(n \times n)$–matrices over $\mathcal{R}$

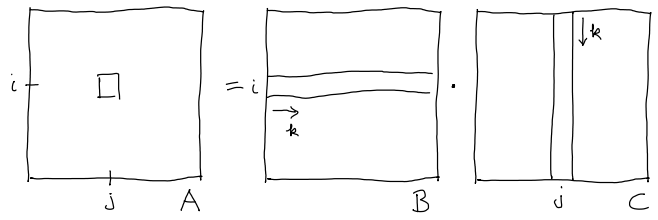
$M_n$ with the following standard operations forms a ring:

- Matrix Addition $A = B+C$: $\forall i, j : \ a_{ij} := b_{ij}+c_{ij}$
- Matrix Multiplication $A = B \cdot C$:

$$\forall i, j : \ a_{ij} := \sum_{k=1}^{n} b_{ik} \cdot c_{kj}$$

- $\mathbf{0}$ element: $\mathbf{0}_{ij} = 0$
- $\mathbf{1}$ element: $\mathbf{1}_{ij} = \delta_{ij}$
  (Kronecker delta: 1 if $i = j$, 0 otherwise)
  Matrix with ones on the main diagonal and zeros everywhere else.






The standard method for multiplying $(n \times n)$–matrices requires the following operations:

$n^2 \cdot (n$ multiplications and $n - 1$ additions over $\mathcal{R})$

$\rightsquigarrow$

$n^3$ multiplications and $n^3 - n^2$ additions over $\mathcal{R}$

What is the smallest number $\omega$ such that two $(n \times n)$–matrices can be multiplied with $O(n^\omega)$ operations?

Clearly $w \geq 2$. Standard method shows $\omega \leq 3$

Breakthrough: Strassen 1969: $\omega \leq 2.81$

Coppersmith & Winograd 1989: $\omega \leq 2.38$
(impractical)

Theorem: (Strassen 1969)

The product of two $(n \times n)$–matrices over a ring can be computed with $O(n^{\log 7})$ ring operations.

This is an improvement from $\Theta(n^3) = \Theta(n^{\log 8})$ to $O(n^{\log 7}) \approx O(n^{2.81})$

We use a Lemma to prove the Theorem:

Lemma: (Winograd)

The product of two $(2 \times 2)$–matrices over a ring can be computed with 7 ring multiplications and 15 ring additions/subtractions.

Remark:

Strassen presented a solution using 7/18 ring operations which leads to the same asymptotic runtime.

Winograd also proved that 7/15 ring operations are optimal.



Proof of Lemma: For

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

$c_{ij}$ can be computed as follows:

$$\begin{array}{ll} s_1 = a_{21} + a_{22} & t_1 = b_{12} - b_{11} \\ s_2 = s_1 - a_{11} & t_2 = b_{22} - t_1 \\ s_3 = a_{11} - a_{21} & t_3 = b_{22} - b_{12} \\ s_4 = a_{12} - s_2 & t_4 = b_{21} - t_2 \end{array}$$

$$\begin{array}{ll} p_1 = a_{11} \cdot b_{11} & p_2 = a_{12} \cdot b_{21} \\ p_3 = s_1 \cdot t_1 & p_4 = s_2 \cdot t_2 \\ p_5 = s_3 \cdot t_3 & p_6 = s_4 \cdot b_{22} \\ p_7 = a_{22} \cdot t_4 & \end{array}$$

$$\begin{array}{ll} c_{11} = p_1 + p_2 & u_2 = p_1 + p_4 \\ u_3 = u_2 + p_5 & c_{21} = u_3 + p_7 \\ c_{22} = u_3 + p_3 & u_6 = u_2 + p_3 \\ c_{12} = u_6 + p_6 & \end{array}$$

7 multiplications and 15 additions/subtractions



Example:

$c_{21} = u_3 + p_7 = u_2 + p_3 + a_{22} \cdot t_4 = u_2 + p_5 + s_1 \cdot t_1 + a_{22} \cdot (b_{21} - t_2)$

$= p_1 + p_4 + s_3 \cdot t_3 + (a_{21} + a_{22}) \cdot (b_{12} - b_{11}) + a_{22} \cdot (b_{21} - b_{22} + t_1)$

$= a_{11} \cdot b_{11} + s_2 \cdot t_2 + (a_{11} - a_{21}) \cdot (b_{22} - b_{12}) + (a_{21} + a_{22}) \cdot (b_{12} - b_{11}) + a_{22} \cdot (b_{21} - b_{22} + b_{12} - b_{11})$

$= a_{11} \cdot b_{11} + (s_1 - a_{11}) \cdot (b_{22} - t_1) + (a_{11} - a_{21}) \cdot (b_{22} - b_{12}) + (a_{21} + a_{22}) \cdot (b_{12} - b_{11}) + a_{22} \cdot (b_{21} - b_{22} + b_{12} - b_{11})$

$= a_{11} \cdot b_{11} + (a_{21} + a_{22} - a_{11}) \cdot (b_{22} - b_{21} + b_{11}) + (a_{11} - a_{21}) \cdot (b_{22} - b_{12}) + (a_{21} + a_{22}) \cdot (b_{12} - b_{11}) + a_{22} \cdot (b_{21} - b_{22} + b_{12} - b_{11})$

$= a_{21} b_{11} + a_{22} b_{21}$



Proof of Theorem:

For $M(n) :=$ number of ring operations for the multiplication of two $(n \times n)$–matrices we have to prove:

$$M(n) \in O(n^{\log 7})$$

First we consider $n = 2^k$:

$$A = \begin{pmatrix} A^{11} & A^{12} \\ A^{21} & A^{22} \end{pmatrix} \quad B = \begin{pmatrix} B^{11} & B^{12} \\ B^{21} & B^{22} \end{pmatrix},$$

where $A^{ij}, B^{ij}$ are $(\frac{n}{2} \times \frac{n}{2})$–matrices. Then

$$A \cdot B = \begin{pmatrix} C^{11} & C^{12} \\ C^{21} & C^{22} \end{pmatrix}, \text{ with}$$

$$C^{11} = A^{11}B^{11} + A^{12}B^{21},$$
$$C^{12} = A^{11}B^{12} + A^{12}B^{22},$$
$$\ldots$$

which can be verified by splitting up the sums for computing entries $(i,j)$ of matrix $C^{ab}$ into two parts. E.g.

$C_{ij} = C^{11}_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}, \quad (1 \le i,j \le \frac{n}{2})$

$= \sum_{k=1}^{\frac{n}{2}} A_{ik}B_{kj} \; + \; \sum_{k=1}^{\frac{n}{2}} A_{i,k+\frac{n}{2}}B_{k+\frac{n}{2},j}$

$= (A^{11} \cdot B^{11})_{ij} \; + \; (A^{12} \cdot B^{21})_{ij}$

Thus, $C^{11} = A^{11}B^{11} + A^{12}B^{21}$.

Because the set of $(\frac{n}{2} \times \frac{n}{2})$–matrices forms a ring, the Lemma can be applied and the $C^{ij}$ can be determined with 7 multiplications and 15 additions/subtractions of $(\frac{n}{2} \times \frac{n}{2})$–matrices.

Therefore:

$$M(n) = 7 \cdot M(\frac{n}{2}) + 15(\frac{n}{2})^2 \quad (n > 1); \quad M(1) = 1$$

Using the master theorem (case 1), we obtain

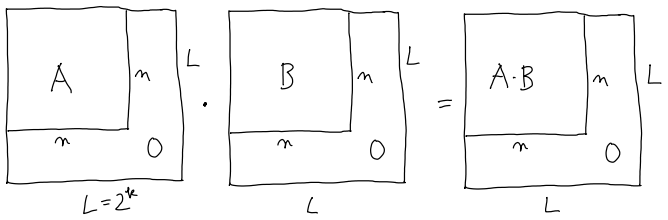
$$M(n) \in \Theta(n^{\log 7})$$

A detailed analysis shows: $M(n) = 6 \cdot n^{\log 7} - 5 \cdot n^2$



If $n$ is not a power of 2 we embed the $(n \times n)$–matrices in $(L \times L)$–matrices — where $L = 2^k$ is the smallest power of 2 larger than $n$ — and compute the product. This technique is called "padding". Lecture 14

Thus, $2^{k-1} < n < 2^k = L$, and therefore $L < 2n$.




Because $M(L) \in O(L^{\log 7})$, we know $M(L) \le cL^{\log 7}$ for a suitable $c$ and large enough $L$.

So, with $L < 2n$:

$$M(n) = M(L) \le c \cdot L^{\log 7} < c \cdot (2 \cdot n)^{\log 7} \in O(n^{\log 7})$$

□

The lowest upper bound is still open. Strassen's $O(n^{\log 7})$ algorithm is currently the only known method faster than the $\Theta(n^3)$ standard algorithm in practice.



## Multiplication of Large Integers

Suppose we are dealing with large integers that are represented by thousands of bits, like so:

$$a = \sum_{i=0}^{n-1} a_i \cdot 2^i,$$

where $a_i \in \{0,1\}$ is the $i$-th digit in the binary representation of $a$. E.g. $18$ (base $10$) $= 10010$ (base $2$)

Such integers may be too big to fit into one machine word (usually 32 or 64 bits). So we need to design an algorithm for multiplying arbitrarily large numbers.

The straightforward "school" method for adding two $n$-bit numbers takes $\Theta(n)$ steps: add corresponding pairs of bits from right to left taking into account carries from the previous addition).

For multiplication, the standard algorithm takes $\Theta(n^2)$ steps (add $n$ numbers). [ Demonstration on board ]

Goal: time $o(n^2)$.

Suppose that $I$ and $J$ are the two $n = 2^k$ bit integers to be multiplied. They can be represented as follows:



$I = A \cdot 2^{\frac{n}{2}} + B$ and $J = C \cdot 2^{\frac{n}{2}} + D$, where $A, B, C,$ and $D$ are $\frac{n}{2}$-bit numbers.

```
          n/2 bits     n/2 bits
        --------------------------
I = |     A      |     B      |
        --------------------------
J = |     C      |     D      |
        --------------------------
```

Then $I \cdot J = AC \cdot 2^n + (AD + BC) \cdot 2^{\frac{n}{2}} + BD$.

Multiplying by $2^n$ and $2^{\frac{n}{2}}$ takes linear time (left shift of bit sequence), and so does adding the results, for a total linear runtime $l_1(n)$ after the 4 multiplications:

$$AC, \quad AD, \quad BC, \quad BD$$

So, the time required for multiplying $I$ and $J$ is:

$$T(n) = 4T(\frac{n}{2}) + l_1(n)$$

Master Theorem: $T(n) \in \Theta(n^2)$.

This it no better than the standard algorithm. The bottleneck is too many recursive calls. So we try to reduce the number of $\frac{n}{2}$ bit multiplications.

Consider:
$R = (A + B)(C + D) = AC + (AD + BC) + BD$

$R$ contains all four terms for computing $I \cdot J$, but we need to separate them. Suppose we computed $P = AC$ and $Q = BD$ as well. Then:

- $(AD + BC) = R - P - Q$
- $AC = P$
- $BD = Q$

This saves us one multiplication. Because computing $R$ is done by multiplying two $(\frac{n}{2} + 1)$-bit numbers, the recurrence now is:

$$T(n) = T(\frac{n}{2} + 1) + 2T(\frac{n}{2}) + l_2(n)$$

for a suitable linear function $l_2$. Not quite in the form the Master Theorem covers.

To get rid of $T(\frac{n}{2}+1)$ we apply the recursion idea once more by setting:

$$I' = A + B \text{ and } J' = C + D$$

and splitting them up in $\frac{n}{2}$-bit numbers $A', C'$ and one-



bit numbers $B', D'$:

$$I' = A'2 + B' \text{ and } J' = C'2 + D'$$

Now:

$$I'J' = (A + B)(C + D) = (A'2 + B')(C'2 + D')$$
$$= A'C'4 + (A'D' + B'C')2 + B'D'$$

Therefore, $T(\frac{n}{2} + 1) = T(\frac{n}{2}) + l_3(n)$ because any multiplication other than $A'C'$ is trivial ($\cdot 0$ or $\cdot 1$). In total:

$$T(n) = 3T(\frac{n}{2}) + l_4(n)$$

($l_4$ linear). Using the Master Theorem and generalizing the result to arbitrary $n$ by a padding argument similar to that for matrix multiplication, we can prove:

Theorem: (Karatsuba and Ofman, 1963)

Two $n$ bit integers can be multiplied in time $\Theta(n^{\log 3})$, where $\log 3 \approx 1.585$.

This is substantially faster than the standard $\Theta(n^2)$ time method. Even faster is the Schönhage–Strassen algorithm: $\Theta(n \log(n) \log\log(n))$.



## MergeSort

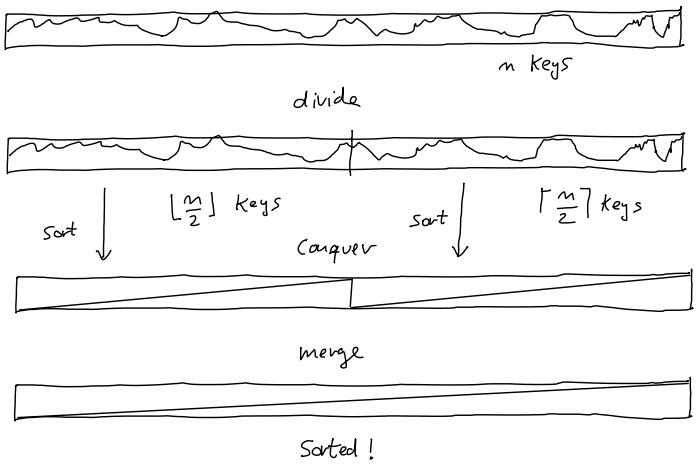
Lecture 15

We are trying to speed up sorting to make it applicable to much larger data sets. The way selection sort works is to decrease the size of the remaining array to sort by one at a time, always moving the minimal key to the front.

How else could we create subarrays to be sorted and put together the results to sort the entire array? We could try to split the array into two equally sized subarrays — sort those — and contruct the whole sorted sequence by "merging" the sorted subarrays — hoping that this can be done quickly.



Illustration



Merge Example:

```
A[] = 3 0 4 7 2 1 6 5

A1[] = 3 0 4 7     A2[] = 2 1 6 5
sort               sort
A1[] = 0 3 4 7     A2[] = 1 2 5 6

merge A1[] and A2[] in non-decreasing order.

A[] =  0 [from A1]
       1 [from A2]
       2 [from A2]
       3 [from A1]
       4 [from A1]
       5 [from A2]
       6 [from A2]
       7 [from A1]
```

How does merging work exactly?

We can work from left to right in both subarrays, because we know the smallest values are stored at the front.

Which number goes first into A[]? 0, because it is the mininum of A1[0] and A2[0]. We copy A1[0]



(=0) to A[0] as the first key and advance i, because we consumed the 0. Now A1[1] gets compared with A2[0]. A2[j] is the smallest, we store it in A[], and advance j. Etc. until we reach the end in both subarrays.

```
A1[] = 0 3 4 7     A2[] = 1 2 5 6
       i                  j         store A1[0]  0
         i                j         store A2[0]  1
         i                  j       store A2[1]  2
         i                    j     store A1[1]  3
           i                  j     store A1[2]  4
             i                j     store A2[2]  5
             i                  j   store A2[3]  6
             i                    j store A1[3]  7
```

In each step either A1[i] or A2[j] is the value to be stored next, we just have to choose the smaller one to go first.

Pseudo code for this operation follows. We assume that A1[] and A2[] are consecutive subarrays of the original array A[] and that we have access to a temporary array C[] which has the same size as A[].



```
// input:
//   A: array containing two adjacent sorted subsequences
//   i1: index of the first key of the first subarray,
//   n1 = length of first subarray
//   n2 = length of second subarray
//   (the second subsequence starts at i1+n1)
//   C: array with the same size as A (scratch space)
//
// task:
//   merge both subsequences in increasing order
//
//
//  n1 = 4, n2 = 5
//               first    second  subarray
//                _____    ______
//               /     \  /      \
//  A[] =  ..... 0 3 4 7 1 2 5 6 2 ....
//               |       |         |
//               i1    i1+n1=endi  |
//               i       |         |
//                       |         |
//                       j       endj = j+n2
//  output:
//
//  A[] =  ..... 0 1 2 3 4 5 6 7 ....
//
// using
//  C[]    ..... ? ? ? ? ? ? ? ? ....
//
// .... parts unchanged
```



```
function Merge(A[], i1, n1, n2, C[])
i    <- i1
endi <- i + n1  // first subarray:  i..endi-1
j    <- endi
endj <- j + n2  // second subarray: j..endj-1
k    <- i       // index into C[]

// as long as there are keys in both subsequences ...
while i < endi AND j < endj do
  if A[i] < A[j] then // store the minimal value into C
    C[k] <- A[i] ; i <- i + 1
  else
    C[k] <- A[j] ; j <- j + 1
  end
  k <- k + 1
end

// here we are done with one of the subarrays or both;
// now simply copy the remaining keys over
while i < endi do // first subarray not finished?
  C[k] <- A[i] ; i <- i + 1 ; k <- k + 1
end
while j < endj do // second subarray not finished?
  C[k] <- A[j] ; j <- j + 1 ; k <- k + 1
end

// copy the sorted keys from C back to A
k <- i1
while k < endj do
  A[k] <- C[k] ; k <- k + 1
end
```

What is the runtime of `Merge`?

We have 4 while loops

We count the number of loop iterations $I(\texttt{n1}, \texttt{n2})$. Then the total runtime is $\Theta(I(\texttt{n1}, \texttt{n2}))$.

In the first 3 loops each iteration either `i` or `j` is incremented by one until they reach `endi` or `endj`, respectively.

This means the total number of iterations there is `n1 + n2`.

The last loop is executed exactly `n1+n2` times.

This means $I(\texttt{n1}, \texttt{n2}) = 2(\texttt{n1} + \texttt{n2})$.

This is good — asymptotically optimal in fact, because we need to visit each of the `n1+n2` keys in any case.



Now that we know how to merge subarrays, let's return to sorting. We want to split the array in two parts, sort those, and merge the resulting subarrays. For sorting the subarrays we call our sorting function recursively.

```
// sort A[i..i+n-1] using C[] as scratch space
// initial call: MergeSort(A[], 0, n, C[])
// to sort A[0..n-1]

function MergeSort(A[], i, n, C[])
if n <= 1 then           // (*)
  return                 // nothing to do
end

// divide
l <- floor(n/2)          // length of left half
r <- n-l                 // length or right half

// conquer
MergeSort(A, i,   l, C)  // sort left subarray
MergeSort(A, i+l, r, C)  // sort right subarray

// combine
Merge(A, i, l, r, C)     // merge subarrays
```



Example: $n = 8$

```
                3 1 8 4 7 2 5 6
divide
          3 1 8 4         7 2 5 6
divide
        3 1     8 4     7 2     5 6
divide
      3   1   8   4   7   2   5   6
merge
        1 3     4 8     2 7     5 6
merge
          1 3 4 8         2 5 6 7
merge
                1 2 3 4 5 6 7 8
```

Runtime

Let $T(n)$ be the total number of loop iterations in function `Merge` plus the number of times line `(*)` is executed. Then $T(1) = 1$ and

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 2(\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil) + 1$$



$$= T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n + 1$$

Applying the Master Theorem yields $T(n) = \Theta(n \log n)$.

So the total runtime of `MergeSort` is $\Theta(n \log n)$, which is significantly faster than `SelectionSort`.

Here is a table that illustrates the improvement of `MergeSort` over `SelectionSort` in terms of key comparisons:

| $n$ | $n \log n$ | $n^2/2$ | MergeSort speed up |
|---|---|---|---|
| $1024$ | $10240$ | $524288$ | $51$ |
| $10^6$ | $20 \cdot 10^6$ | $0.5 \cdot 10^{12}$ | $25086$ |
| $10^9$ | $23 \cdot 10^9$ | $0.5 \cdot 10^{18}$ | $21 \cdot 10^6$ |

A disadvantage of `MergeSort` is that it needs additional space of size $n$. An advantage is that `MergeSort` accesses keys in succession, which makes it suitable for sorting data stored in files.

## QuickSort

Lecture 16

The idea:

- Pick a key x
- Partition array into two subarrays that contain keys $\leq$ or $\geq$ x, respectively
- Recursively sort the subarrays

```
// sorts A[l]..A[r]
function QuickSort(A[l..r])
if l < r then
  i <- Partition(A[l..r])
  QuickSort(A[l..i-1])
  QuickSort(A[i+1..r])
end
```



If the following properties hold for `Partition`, it is easy to show that `QuickSort` sorts correctly:

- Key `A[i]` is at its final place when `A[l..r]` is sorted
- `A[l]..A[i-1]` $\leq$ `A[i]`
- `A[i+1]..A[r]` $\geq$ `A[i]`

Idea for partition:

```
| | | | | | | | | | | |x|
 l     i|       |j       r
        |       |
  <= x  |  > x  |    ?
```

- Pick pivot key $x$ =`A[r]`
- Visit all keys from left to right and maintain location $i+1$ for next key $\leq x$
- When key $\leq x$ is found swap with location $i+1$
- At end, swap key $i+1$ with key $r$



```
function Partition(A[l..r])
x <- A[r] // pivot key
i <- l-1
for j <- l to r-1 do
  if A[j] <= x then    // (*)
    i <- i + 1
    swap A[i] and A[j]
  end
end
swap A[i+1] and A[r]   // (**)
return i+1
```

```
  2 8 7 1 3 5 6 4   at (*)
i j                 swap
  2 8 7 1 3 5 6 4
  i j                no
  2 8 7 1 3 5 6 4
  i   j              no
  2 8 7 1 3 5 6 4
  i     j           swap
  2 1 7 8 3 5 6 4
    i     j         swap
  2 1 3 8 7 5 6 4
      i     j        no
  2 1 3 8 7 5 6 4
      i       j      no
  2 1 3 8 7 5 6 4   done, final swap (**)
  2 1 3 4 7 5 6 8
```



Partition Correctness and Runtime

Loop invariant at `(*)`:

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

```
| | | | | | | | | | | |x|
 l     i|       |j       r
        |       |
  <= x  |  > x  |    ?
```

Initialization: $i = l-1$ and $j = l$. The two ranges in 1. and 2. are empty, so the conditions are trivially satisfied. 3. holds because $x = A[r]$.

Maintenance: 2 cases: if $A[k] > x$ then only $j$ is incremented and condition 2. again holds and all entries are unchanged. If $A[k] \leq x$ then $A[i+1]$ and $A[j]$ are swapped and $i$ and $j$ are incremented, at which point the L.I. again holds.

Termination: At termination, $j = r$. Therefore, values are partitioned into three sets: $\leq x$, $> x$, $= x$, and

location $i+1$ is valid for $A[r]$.

`Partition` executes $n-1$ key comparisons where $n$ is the number of keys ($\texttt{r}-\texttt{l}+1$).

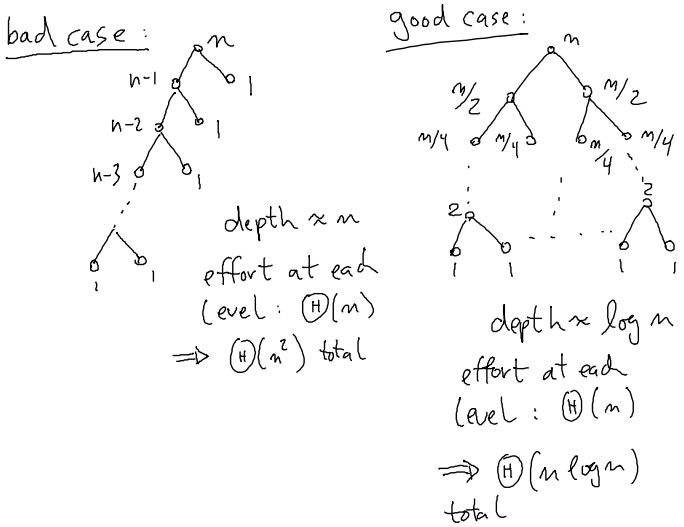
Worst Case Runtime Analysis

Theorem: In the worst case QuickSort needs $\Theta(n^2)$ comparisons for inputs of size $n$.

Proof: Let $T(n)$ be the worst case number of comparisons for input size $n$

Intuition: $T(n)$ high when partition split is uneven






Consider sorted array `A[0..n-1]`: $n-1$ comparisons, then $n-2$ comparisons, etc. Therefore,
$T(n) \geq (n-1)+(n-2)+\cdots+1 = (n-1)n/2$, and $T(n) \in \Omega(n^2)$



For establishing the upper bound we note

$$T(n) \leq \max_{0 \leq q \leq n-1} (T(q) + T(n-1-q)) + n - 1$$

and $T(0) = T(1) = 0$. We guess $T(n) \leq c \cdot n^2$ for some $c > 0$ and all $n \in \mathbb{N}$. Using the induction hypothesis which we assume to hold for $q < n$ for $n \geq 2$ we obtain

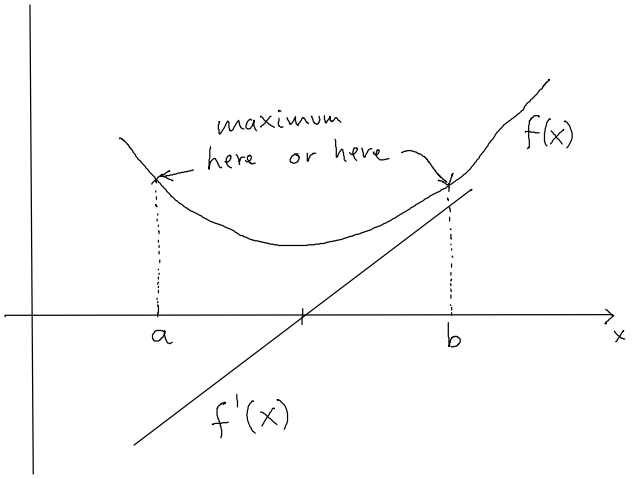
$$T(n) \leq \max_{0 \leq q \leq n-1} (cq^2 + c(n-1-q)^2) + n - 1$$

$$= c \max_{0 \leq q \leq n-1} \underbrace{(q^2 + (n-1-q)^2)}_{\text{quadratic function in } q} + n - 1$$

What is the maximum value of this function?



[ Calculus: if $f(x)$ is differentiable twice over $[a,b]$ with $f''(x) \geq 0$ (which means the derivative of $f$ is monotonically increasing and $f$ is convex) then $\max_{x \in [a,b]} f(x) = f(a)$ of $f(b)$]




Thus,

$$T(n) \leq c(n-1)^2 + n - 1$$
$$= cn^2 - 2cn + 1 + n - 1 \;\leq\; cn^2$$

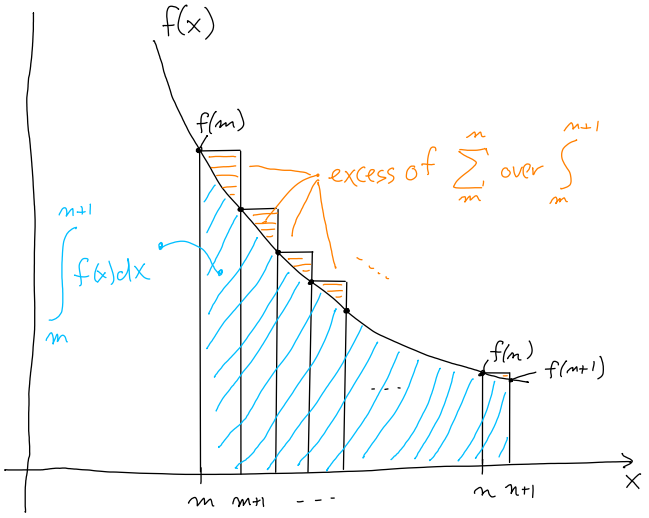
if $c = 1$ and $n \geq 1$. Thus, $T(n) \in \Theta(n^2)$. □

## Bounding Sums Lecture 17

Before we analyse the average case runtime of `Quick-Sort`, we present useful tools for bounding sums.

Calculus: if $f$ is monotonically decreasing, then

$$\sum_{i=m}^{n} f(i) \;\geq\; \int_{m}^{n+1} f(x)\,dx$$






Similarly: if $f$ is monotonically decreasing, then

$$\sum_{i=m}^{n} f(i) \;\leq\; \int_{m-1}^{n} f(x)\,dx$$

Analogous bounds hold for monotonically increasing functions $f$.

Example: We want to find tight bounds for the $n$-th Harmonic Number $H_n := \sum_{i=1}^{n} \frac{1}{i}$.

The anti-derivative of $\frac{1}{x}$ is $\ln x$ and $\frac{1}{x}$ is monotonically decreasing. Therefore:

$$\sum_{i=2}^{n} \frac{1}{i} \;\leq\; \int_{1}^{n} \frac{1}{x}\,dx \;=\; \ln x\,|_1^n \;=\; \ln n$$

$$\rightsquigarrow\; H_n \;=\; 1 + \sum_{i=2}^{n} \frac{1}{i} \;\leq\; 1 + \ln n$$

Likewise,

$$\sum_{i=1}^{n} \frac{1}{i} \;\geq\; \int_{1}^{n+1} \frac{1}{x}\,dx \;=\; \ln x\,|_1^{n+1} \;=\; \ln(n+1)$$



Putting both inequalities together we get:

$$\forall n > 0: \;\; \ln n \leq \ln(n+1) \;\leq\; H_n \;\leq\; 1 + \ln n$$

Tight approximation! $H_n \approx \ln n$ (ratio limit 1)

Even better: $\lim_{n\to\infty} H_n - \ln n = \gamma \doteq 0.5772...$
(Euler-Mascheroni constant)

More tools for bounding sums:

$$\sum_{i=m}^{n} f(i) \;\leq\; (n-m+1)\cdot \max\{f(i) \mid i = m..n\}$$

$$\sum_{i=m}^{n} f(i) \;\geq\; (n-m+1)\cdot \min\{f(i) \mid i = m..n\}$$

Example: $\sum_{i=1}^{n} i \;\leq\; n^2$

If $f \geq 0$ and monotonically increasing, then

$$\sum_{i=1}^{n} f(i) \;\geq\; \sum_{i=m}^{n} f(i) \;\geq\; (n-m+1)\cdot f(m)$$



Example 1:

$$\sum_{i=1}^{n} i \;\geq\; \sum_{i=\frac{n}{2}}^{n} i \;\geq\; (n - \frac{n}{2} + 1)\frac{n}{2} = \frac{n^2}{4} + \frac{n}{2}$$

($n$ even, inequality also holds for odd $n$, exercise)

$$\rightsquigarrow\; \sum_{i=1}^{n} i \;\in\; \Theta(n^2)$$

Example 2:

$$\sum_{i=1}^{n} \log i \;\leq\; n \log n$$

$$\sum_{i=1}^{n} \log i \;\geq\; \sum_{i=\frac{n}{2}}^{n} \log i \;\geq\; (\frac{n}{2}+1)\log(\frac{n}{2})$$

$$= \frac{n}{2}\log(n) - (\frac{n}{2}+1), \quad n \text{ even}$$

Similar bound for $n$ odd (exercise).

$$\rightsquigarrow\; \sum_{i=1}^{n} \log i \;\in \Theta(n \log n)$$

## QuickSort Average Case Analysis Lecture 18

Theorem: Assuming all $n$ keys are distinct and all $n!$ input sequences are equally probable, the average number of key comparisons in `QuickSort` is $\approx 2n \ln n$.

Proof Sketch:

Let $A(n)$ be the average number of key comparisons for input size $n$.

Facts:

- Each key is equally likely to become the pivot key.
- After calling `Partition` all subsequences are equally probable. This can be verified by counting sequences that lead to the same subsequences.

Then $A(0) = A(1) = 0$, and

$$A(n) = n - 1 + \frac{1}{n}\sum_{k=1}^{n}(A(k-1) + A(n-k)),\ n \geq 2$$

[ add the average runtime for each of the $n$ cases and divide by $n$ to get the average, then add $n-1$ `Partition` key comparisons ]



Goal: transform this recurrence into form

$$B(n) = B(n-1) + f(n)$$

which we hopefully can handle by iterated substitution.

$$A(n) = n - 1 + \frac{1}{n}\sum_{k=1}^{n}(A(k-1) + A(n-k))$$

[ all $A$ terms appear twice ]

$$A(n) = n - 1 + \frac{2}{n}\sum_{k=1}^{n} A(k-1) \quad |\cdot n$$

$$nA(n) = n(n-1) + 2\sum_{k=1}^{n} A(k-1) \quad |-(n-1)A(n-1)$$

[ get rid of sum by subtracting "$(n-1)$-case" from "$n$-case", now assuming $n \geq 3$ ]

$$\begin{aligned} &nA(n) - (n-1)A(n-1) \\ &= n(n-1) - (n-1)(n-2) + 2A(n-1) \\ &nA(n) = (n+1)A(n-1) + 2n - 2 \quad |\div n(n+1) \end{aligned}$$

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2n-2}{n(n+1)} = \frac{A(n-1)}{n} + \frac{4}{n+1} - \frac{2}{n}$$



which has the form we aimed for. Iterating $i$ times ...

$$= \frac{A(n-i)}{n-i+1} + \sum_{j=1}^{i}\frac{4}{n+2-j} - \sum_{j=1}^{i}\frac{2}{n+1-j}$$

[ base case $2$ is reached for $i = n-2$ ]

$$= \frac{A(2)}{3} + \sum_{j=1}^{n-2}\frac{4}{n+2-j} - \sum_{j=1}^{n-2}\frac{2}{n+1-j}$$

[ $A(2) = 3$, denominators step from $n+1$ down to $4$ and from $n$ down to $3$, respectively ]

$$= 1 + \sum_{k=4}^{n+1}\frac{4}{k} - \sum_{k=3}^{n}\frac{2}{k} = 1 + 4\sum_{k=4}^{n+1}\frac{1}{k} - 2\sum_{k=3}^{n}\frac{1}{k}$$

$$= 1 + 4(H_n + \frac{1}{n+1} - H_3) - 2(H_n - H_2)$$

$$= 2H_n + \frac{4}{n+1} - C \approx 2\ln n$$

In summary:

$$\frac{A(n)}{n+1} \approx 2\ln n$$

$$A(n) \approx 2n\ln n \doteq 1.38 n \log n \qquad \square$$



## QuickSort Improvements

– There are input sequences for which QuickSort needs quadratic runtime. However, if the input order is randomized before starting QuickSort, the expected runtime is $\Theta(n \log n)$ for any fixed input with distinct keys:

```
Randomize(A[0..n-1])
QuickSort(A[0..n-1])
```

– Use InsertionSort for small subsequences:

```
// M = 5..25 architecture dependent
if r-l < M then
  InsertionSort(A[l..r])
  return
end
```

– reduce worst case memory overhead from $\Theta(n)$ for call-stack to $\Theta(\log n)$ by removing tail-recursion (= function call at the end of a function) by a while loop that calls QuickSort only once for the smaller sub-problem in its body and instead of the second call just adjusts `l` and `r`).

Lecture 19

– Pick better pivot key. Best case: median. Not easy to find ($\Theta(n)$ time, but complex and large constant).

Approximation: median-of-three key:

- sort `A[l], A[m], A[r] (m = l+(r-l)/2)`

- exchange `A[m], A[r-1]`

- call `Partition(A[l+1..r-1])`

This is generally faster, but $\Theta(n^2)$ worst case time remains.

In addition, none of the above improvements significantly speed up truly degenerate cases such as $n$ equal keys.



## Pseudo Random Number Generators (PRNGs)

Randomness is a powerful resource!

Randomized QuickSort fast on average for any input with distinct keys

Many other fast randomized algorithms exist.

How do we generate "random" numbers?

No such thing when using standard computer hardware.

Using programs to generate numbers $\rightsquigarrow$ may look like, BUT ARE NOT RANDOM

Want fast algorithm that creates sequences of numbers that pass basic statistical tests (such as having a certain mean, variance, correlation).

Popular are Linear Congruential Generators (LCGs) of the form:

$$X_0 = \text{"seed"}$$
$$X_{t+1} = (a \cdot X_t + c) \bmod m$$

for constants $a, c, m$. Typical $m$ are powers of 2 because we use binary computers on which $\bmod 2^k$ is



cheap (why?).

Examples: $a = 1, c = 1, m = 2^{32}$ :
$X_0 = 0, X_1 = 1, X_2 = 2, \ldots$

Successors are very correlated.

$a = 1664525, c = 1013904223, m = 2^{32}$

$X_0 = 0, X_1 = 1013904223, X_2 = 1196435762, X_3 = 3519870697, \ldots$

Beware: don't choose constants randomly and don't rely on lower-order bits which might be cycling with small period length.

For above LCG choose $a, c, m$ as follows to achieve maximum period length $m$ for all seed values:

1. $c$ and $m$ are relatively prime, i.e. their only common divisor is 1,
2. $a - 1$ is divisible by all prime factors of $m$,
3. $a - 1$ is a multiple of 4 if $m$ is a multiple of 4.

(more about PRNGs at Wikipedia, thorough treatment: D. Knuth: The Art of Computer Programming)



## Application: Randomize Array for QuickSort

```
// assumes random() returns uniformly distributed
// integer values in {0..RAND_MAX}

function Randomize(A[0..n-1])
for i <- n-1 downto 1
  // pick index uniformly distributed in [0..i]

  // Version 1:
  // this code makes use of higher order bits
  // by first computing a random number in [0,1).
  // This avoids small period lengths

  j <- floor((random()/(RAND_MAX+1.0))*(i+1))

  // Version 2: faster but perhaps less random
  // j <- random() % (i+1)  // % = remainder

  swap A[i] and A[j]
end
```

Exercise: prove that each resulting permutation is equally likely

## Heaps

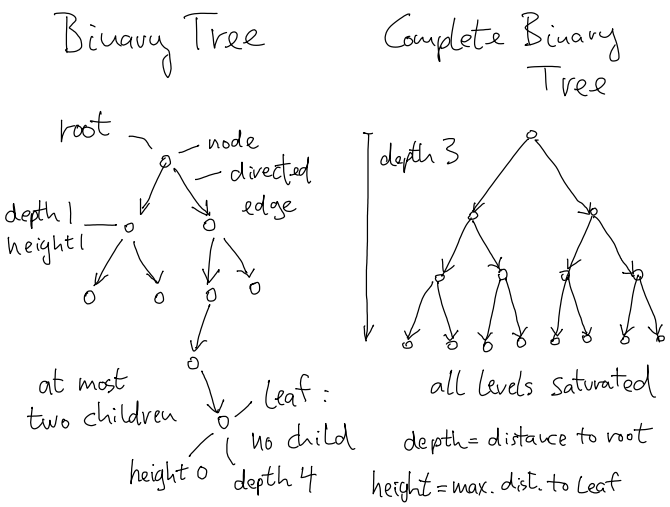
A heap is a data structure that supports the following set of (priority queue) operations:

- Insert($A, n, x$) time $O(\log n)$
- RemoveMax($A, n$) time $O(\log n)$
- Max($A$) time $O(1)$

Heaps can also be used to sort $n$ items in worst case time $O(n \log n)$ in-place: HeapSort (Williams, Floyd 1964), as we will see later.



## Graph theory terminology we need






Let $U$ be totally ordered by $\leq$, i.e. $\leq$ is reflexive ($a \leq a$), transitive ($a \leq b \wedge b \leq c \Rightarrow a \leq c$), and antisymmetric ($a \leq b \wedge b \leq a \Rightarrow a = b$).

Definition: A **heap** is a binary tree with $n$ nodes $\{1..n\}$ that

a) is constructed by removing nodes $n+1, ..., m$ from a complete binary tree with nodes $1, .., m$ in canonical order and

b) in which each node $v$ has a value $A[v] \in U$ that obeys the following order constraint:

$A[v] \geq A[v']$ for all $v$ and children $v'$ of $v$

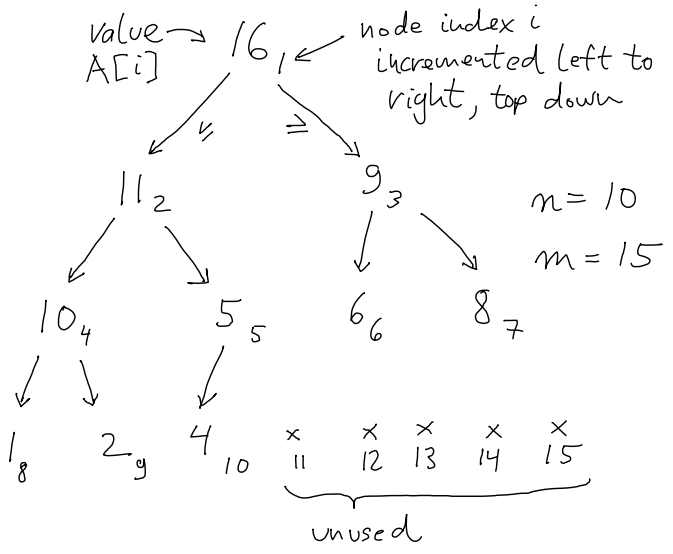
This is called the **Heap Property**.

The heap property ensures that the root value $A[1]$ is the maximum among all values in the heap. This can be shown by induction on the depth of the tree.

If the depth of the tree is logarithmic in $n$, the hope is that operations on heaps, such as removing and adding nodes, is fast.



Example:



Heaps can be represented implicitely by array $A[1 \ldots n]$ without the need of pointers to children like so:

- root has index 1
- left child$(i) = 2i$, if $2i \leq n$
- right child$(i) = 2i + 1$, if $2i + 1 \leq n$
- parent$(i) = \lfloor i/2 \rfloor$ for $i > 1$

Heap property in this representation:

$$A[i] \geq A[2i] \quad \text{for } 1 \leq i \leq n/2$$

and

$$A[i] \geq A[2i+1] \quad \text{for } 1 \leq i < n/2$$



We will discuss implementations of RemoveMax and HeapSort

We make use of two helper functions:

Update$(A, i, n)$ — repairs a heap in time $O(\log n)$

BuildHeap$(A, n)$ — creates a heap using Update in time $O(n)$

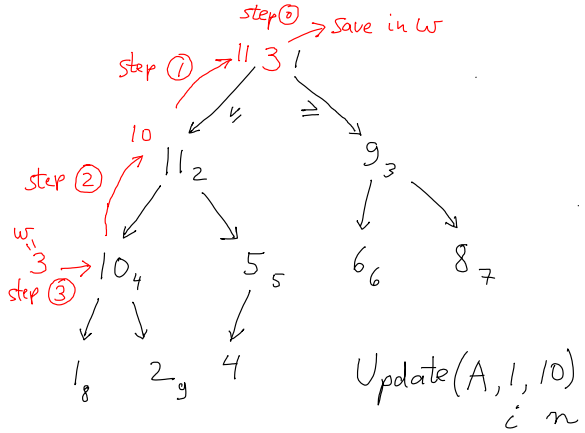


## Update$(A, i, n)$

Pre–condition: in $A[i..n]$ the heap property holds for all nodes except maybe for node $i$

Post–condition: heap property holds for all nodes in $A[i..n]$

Idea: if heap property is violated, repeat exchanging value with largest child






Lecture 20

```
function Update(A,i,n)
w <- A[i]
p <- i      // current index, steps down tree
j <- 2 * p  // left child of p

while j <= n do
  if j < n and A[j] < A[j+1] then j <- j + 1 end
  // j index of maximum child
  if w >= A[j] then
    break            // heap property OK
  end
  // copy value to parent and step down
  A[p] <- A[j]; p <- j; j <- 2 * p
end
// place w at correct position
A[p] <- w
```

## Runtime of Update$(A, i, n)$

The longest $j$ sequence consists of nodes $2^1 i, 2^2 i..2^l i$ (going left all the time) where $l$ is the smallest number with $2^l i > n$, in which case exactly $l-1$ loop iterations take place. Going right produces larger indexes, and so the sequences then can't be longer.

$\Rightarrow \quad n \geq 2^{l-1} i \quad$ ($l$ smallest number with $2^l i > n$)

$\Rightarrow \quad l \leq \lfloor \log(n/i) + 1 \rfloor$

$\Rightarrow$ Update$(A, i, n)$ executes at most $\lfloor \log(n/i) \rfloor$ loop iterations and at most $2 \cdot \lfloor \log(n/i) \rfloor$ key comparisons

Worst case runtime $O(\log(n/i))$

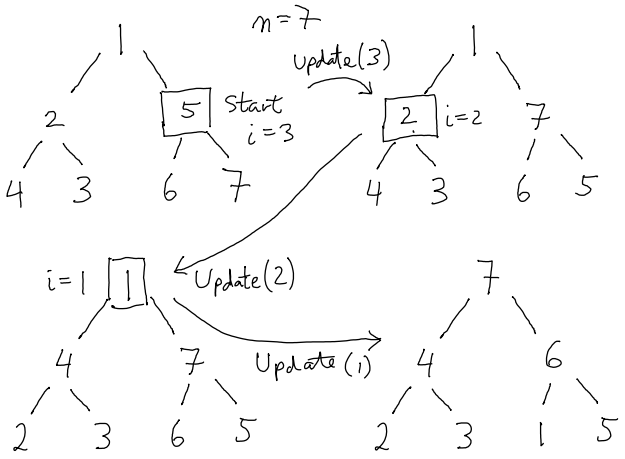


BuildHeap$(A, n)$

How to create a heap using Update?

Idea: use small heaps to build larger ones

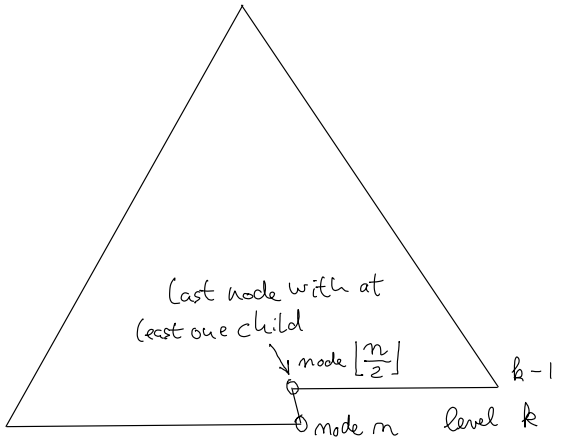
Start close to the leaves and repair sub-heaps for which we know that only their roots can violate the heap property.




By definition, single-node heaps obey the heap property.



So we can skip the last level and start with the rightmost node in the second last level counting down, repairing heaps as we go up.




The rightmost node in the second last level that has a child is the parent of node $n$, i.e. $\lfloor n/2 \rfloor$.

```
function BuildHeap(A, n)
for i <- floor(n/2) downto 1 do
  Update(A,i,n)
end              // Counting down is crucial!
```



Worst case number of comparisons in BuildHeap$(A, n)$:

$$2 \sum_{i=1}^{\lfloor n/2 \rfloor} \lfloor \log(n/i) \rfloor \quad \leq \quad 2 \sum_{i=1}^{\lfloor n/2 \rfloor} (\log n - \log i)$$

$$[ \quad \log(ab) = \log a + \log b \quad ]$$

$$[ \quad \sum_{i=1}^{n} \log i = \log(\prod_{i=1}^{n} i) = \log(n!) \quad ]$$

$$= \quad 2(\lfloor \frac{n}{2} \rfloor \log n - \log(\lfloor \frac{n}{2} \rfloor !)$$

$$\leq \quad 2(\frac{n}{2} \log n - \log(\lfloor \frac{n}{2} \rfloor !)$$

$$[ \quad \log(n!) = n \log n - n \log e + \frac{1}{2} \log n + \Theta(1) \quad ]$$

$$[ \quad \log(\lfloor \frac{n}{2} \rfloor !) \quad \approx \quad \frac{n}{2} \log(\frac{n}{2}) - \frac{n}{2} \log e \quad ]$$

$$\approx \quad (1 + \log e)n \quad \approx \quad 2.44n$$

Runtime proportional to number of comparisions: $O(n)$

The RemoveMax function overwrites the root value with $A[n]$, decreases $n$, and repairs the heap. The current number of nodes $n$ is passed by reference to allow the function to change its value.

```
function RemoveMax(A, ref n)
if n >= 1 then
  A[1] <- A[n]
  n <- n-1
  Update(A, 1, n)
end
```

Summarizing our results thus far:

Theorem:

a) BuildHeap$(A, n)$ transforms $A$ into a heap in time $O(n)$.

b) RemoveMax$(A, n)$ runs in time $O(\log n)$

c) Max$(A)$ runs in time $O(1)$ (just return $A[1]$)



## HeapSort

Starting with a heap structure we can sort an array by iteratively exchanging the current maximum element (stored at the root) with the last element, and repairing the heap with size decremented by 1.

```
function HeapSort(A,n)
BuildHeap(A,n)
for i <- n downto 2 do
  swap A[i] and A[1]
  Update(A,1,i-1)
end
```

The worst case loop runtime is proportional to the total number of comparisons in Update:

$$\leq 2\sum_{i=2}^{n}\lfloor\log(i-1)\rfloor \;\leq\; 2\log((n-1)!)$$
$$\leq\; 2n\log n - 2(\log e)n + O(\log n)$$



The total number of comparisons is at most

$$2n\log n - (\log e - 1)n + O(\log n)$$

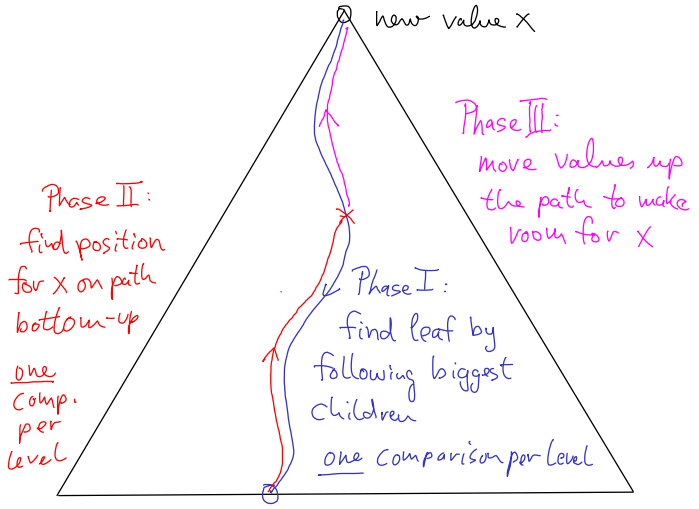
For comparison: the average number of comparisons in QuickSort is $\approx 1.38n\log n$

Can we improve HeapSort so that it beats QS's average number of comparisons?

Yes! Bottom-Up HeapSort + Improvement
(Carlsson 1987, Wegener 1993)



Problem with Update: 2 comparisons per level.

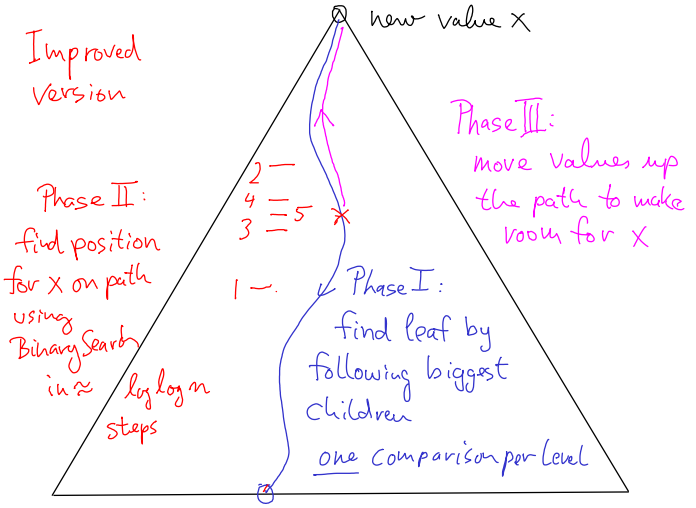



Modify Update$(A, i, n)$ as follows:

1. compute leaf node determined by following the greatest child (1 comparison per level)
2. starting from the leaf find position of $x = A[i]$
3. move every element along the remaining path one level up and store $x$ at its location

Bottom–Up HeapSort performs a linear search in 2. resulting in the worst case number of comparisons of $1.5 \cdot n \log n + O(n)$ (analysis is tricky and based on the fact that many elements sink in deeply)

Improvement: use binary search to determine position of $x$




Problem: How? We search along a path in a heap, not in a linear range!



Observation:

Starting with leaf $l$ the indexes of nodes along the path are

$$l,\ \lfloor l/2 \rfloor,\ \lfloor \lfloor l/2 \rfloor /2 \rfloor = \lfloor l/4 \rfloor,\ \ldots$$

Thus, the index of the $k$-th node up the path from $l$ is $\lfloor l/2^k \rfloor$

Proof: Exercise!

Can be computed quickly. E.g. C code: `l >> k`

Theorem: The improved Update function requires at most

$$\lfloor \log(n/i) \rfloor + \lceil \log \left( \lfloor \log(n/i) \rfloor + 1 \right) \rceil$$

key comparisons (instead of $2\lfloor \log(n/i) \rfloor$). □



With this improvement Bottom–Up HeapSort requires at most

$$\underbrace{(1+\log e)n}_{\text{BuildHeap}} + \sum_{i=2}^{n} (\lfloor \log(i-1) \rfloor + \lceil \log \left( \lfloor \log(i-1) \rfloor + 1 \right) \rceil$$

$$\leq (1+\log e)n + \underbrace{n \log n - (\log e)n + O(\log n)}_{\text{already seen}}$$

$$+ n \log \log(2n) + n$$

$$\leq n \log n + n \log \log(2n) + 2n + O(\log n)$$

comparisons, which is optimal up to lower order terms!

Will see next: Lower bound for number of comparisons:

$$\log n! = n \log n - n \log e + \frac{1}{2} \log n + \Theta(1) \approx n \log n - 1.44n$$



Other sorting algorithms that are optimal w.r.t. the number of comparisons:

- InsertionSort using BinarySearch
  (but $\Theta(n^2)$ assignments in the worst case)
- MergeSort
  (at most $n \log n$ assignments, but linear additional space)

Fast implementations of QuickSort beat HeapSort on typical inputs by a factor of 2 to 5!

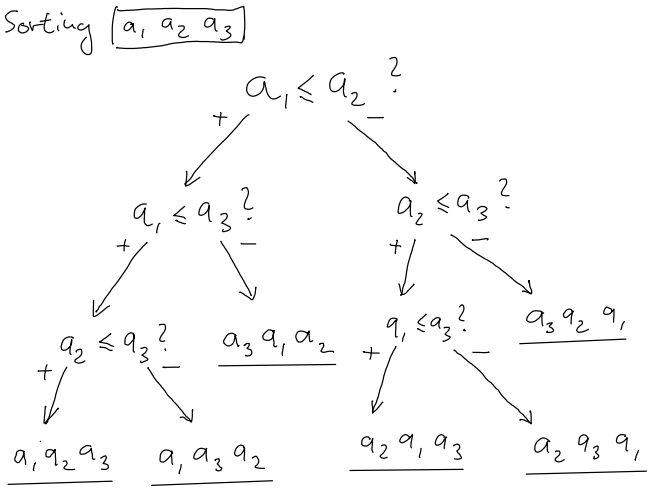
But: QuickSort has quadratic runtime in the worst case, whereas HeapSort's $O(n \log n)$ performance is guaranteed.

Idea: combine both sorting algorithms! (Assignment)

## Runtime Lower Bound for Sorting

Here we establish a lower bound for the number of key comparisons in sorting algorithms that are based on comparing keys.

Given input $(a_1, \ldots, a_n)$, we can view a particular sorting process as a binary decision tree which at its leaves tells us in which order the items would be sorted.

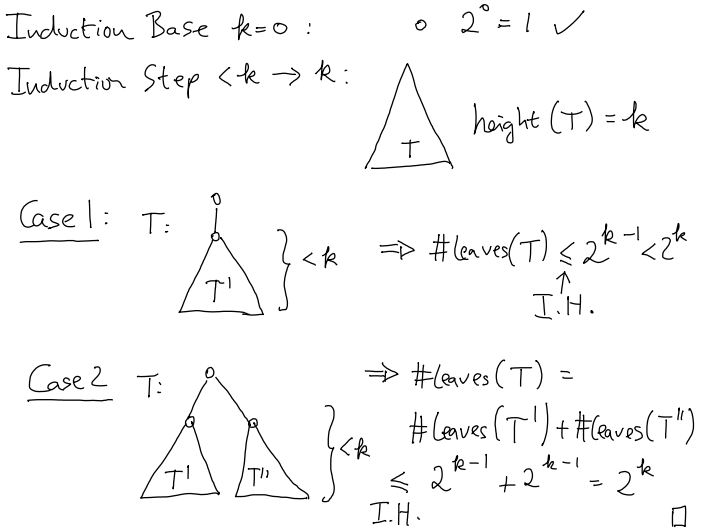





Each permutation of $\{1..n\}$ must be represented by at least one leaf. This binary search tree must have $\geq n!$ leaves, and the worst case number of comparisons is the maximum distance of a leaf to the root (height).

**Lemma:**
Binary trees with height $k$ have at most $2^k$ leaves.

**Proof:** Induction on $k$:




□



**Corollary:**
Any binary tree with $n$ leaves has height $\geq \lceil \log n \rceil$.

**Proof:** Suppose it has height $h \leq \lceil \log n \rceil - 1$.

Then with the Lemma, the number of leaves is

$$\leq 2^h \leq 2^{\lceil \log n \rceil - 1} < 2^{\log n} = n,$$

because $\lceil x \rceil < x + 1$ for all $x \in \mathbb{R}$.

This contradicts the premise that we have $n$ leaves. □

Lecture 21

**Theorem:** Any key comparison based sorting algorithm requires at least $\lceil \log n! \rceil \approx n \log n - 1.44n$ key comparisons for sorting $n$ items in the worst case.

**Proof:** Binary Decision Tree + Corollary

For the approximation we use an earlier result which is based on Stirling's formula for $n!$:

$$\log n! = n \log n - n \log e + \frac{1}{2} \log n + \Theta(1)$$
$$\approx n \log n - 1.44n$$

□



Sorting Runtime Summary

| Algorithm | Worst Case Key Comp. | Average Case Key Comp. |
|---|---|---|
| InsertionSort | $\approx n \log n$ | $\approx n \log n$ (WC-time $\Theta(n^2)$) |
| SelectionSort | $\approx n^2/2$ | $\approx n^2/2$ |
| MergeSort | $\approx n \log n$ | $\approx n \log n$ |
| QuickSort | $\Theta(n^2)$ | $\approx 1.38 n \log n$ |
| B-U HeapSort | $\approx n \log n$ | $\approx n \log n$ |

So, InsertionSort, MergeSort, and B-U HeapSort are asymptotically optimal in terms of key comparisons in the worst case.