

Divide and Conquer

Algorithms

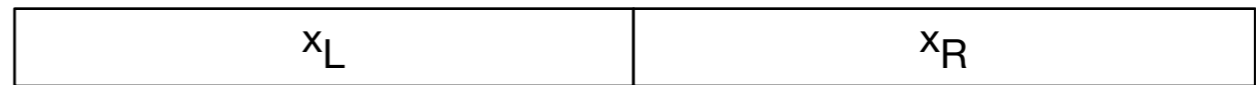
Divide and Conquer

- Generic recipe for many solutions:
 - Divide the problem into two or more smaller instances of the same problem
 - Conquer the smaller instances using recursion (or a base case)
 - Combine the answers to solve the original problem

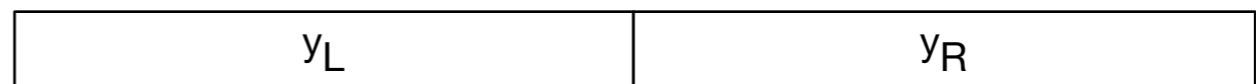
Integer Multiplication

- Assume we want to multiply two n -bit integers with n a power of two
 - Divide: break the integers into two $n/2$ -bit integers

$$x = 2^{\frac{n}{2}}x_L + x_R$$



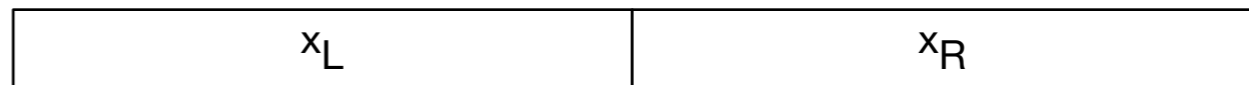
$$y = 2^{\frac{n}{2}}y_L + y_R$$



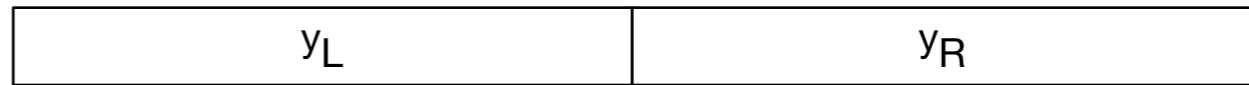
Integer Multiplication

- Conquer: Solve the problem of multiplying of $n/2$ bit integers by recursion or a base case for $n=1$, $n=2$, or $n=4$

$$x = 2^{\frac{n}{2}}x_L + x_R$$



$$y = 2^{\frac{n}{2}}y_L + y_R$$



$$x_L \cdot y_L \quad x_L \cdot y_R \quad x_R \cdot y_L \quad x_R \cdot y_R$$

Integer Multiplication

- Now combine:
 - In the naïve way:

$$\begin{aligned}x \cdot y &= (x_L \cdot 2^{\frac{n}{2}} + x_R) \cdot (y_L \cdot 2^{\frac{n}{2}} + y_R) \\ &= x_L \cdot y_L \cdot 2^n + (x_L \cdot y_R + x_R \cdot y_L) \cdot 2^{\frac{n}{2}} + x_R \cdot y_R\end{aligned}$$

Integer Multiplication

$$\begin{aligned}x \cdot y &= (x_L 2^{\frac{n}{2}} + x_R) \cdot (y_L 2^{\frac{n}{2}} + y_R) \\ &= x_L \cdot y_L 2^n + (x_L \cdot y_R + x_R \cdot y_L) \cdot 2^{\frac{n}{2}} + x_R \cdot y_R\end{aligned}$$

- We count the number of multiplications
 - Multiplying by powers of 2 is just shifting, so they do not count
 - $T(n)$ number of bit multiplications for integers with 2^n bits:
 - Recurrence: $T(0) = 1$
 $T(n + 1) = 4T(n)$

Integer Multiplication

- Solving the recursion

$$T(0) = 1$$

$$T(n + 1) = 4T(n)$$

- Intuition:

$$T(n) = 4T(n - 1) = 4^2T(n - 2) = 4^3T(n - 3) = \dots = 4^nT(0) = 4^n$$

Integer Multiplication

- Proposition: $T(n) = 4^n$
- Proof by induction:
 - Induction base:
$$T(0) = 1 = 4^0$$
 - Induction step: Assume $T(n - 1) = 4^{n-1}$. Show $T(n) = 4^n$
 - Proof: $T(n) = 4T(n - 1)$ Recursion Equation
 - $= 4 \times 4^{n-1}$ Induction Assumption
 - $= 4^n$

Integer Multiplication

- Since the number of bits is $m = 2^n$

- Number of multiplications is

$$S(m) = T(n) = 4^n = (2^n)^2 = m^2$$

- This is not better than normal multiplication

Integer Multiplication

- Now combine:

- Instead:
$$x \cdot y = (x_L 2^{\frac{n}{2}} + x_R) \cdot (y_L 2^{\frac{n}{2}} + y_R)$$
$$= x_L \cdot y_L \cdot 2^n + (x_L \cdot y_R + x_R \cdot y_L) \cdot 2^{\frac{n}{2}} + x_R \cdot y_R$$

- Use $(x_L \cdot y_R + x_R \cdot y_L) = (x_L + x_R) \cdot (y_L + y_R) - x_L \cdot y_L - x_R \cdot y_R$

- This reuses two multiplications that are already used

Integer Multiplication

- We need to deal with the potential overflow in calculating

$$(x_L + x_R) \cdot (y_L + y_R)$$

Integer Multiplication

- Now, we only do three multiplications of 2^n bit numbers in order to multiply two 2^{n+1} bit numbers
- The recursion becomes

$$T(0) = 1 \quad T(n + 1) = 3T(n)$$

Integer Multiplication

- Solving the recurrence $T(0) = 1$ $T(n + 1) = 3T(n)$
- Heuristics:

$$T(n) = 3T(n - 1) = 3^2T(n - 2) = \dots = 3^nT(0) = 3^n$$

Integer Multiplication

- As before prove exactly using induction

Integer Multiplication

- The multiplication of two $m = 2^n$ -bit numbers takes

$$\begin{aligned} S(m) &= T(n) \\ &= 3^n \\ &= 3^{\log_2(m)} \\ &= \exp(\log(3^{\log_2(m)})) \\ &= \exp(\log_2 m \log 3) \\ &= \exp(\log m \log 3 \frac{1}{\log 2}) \\ &= \exp(\log(m^{\log_2 3})) \\ &= m^{\log_2 3} \end{aligned}$$

Integer Multiplication

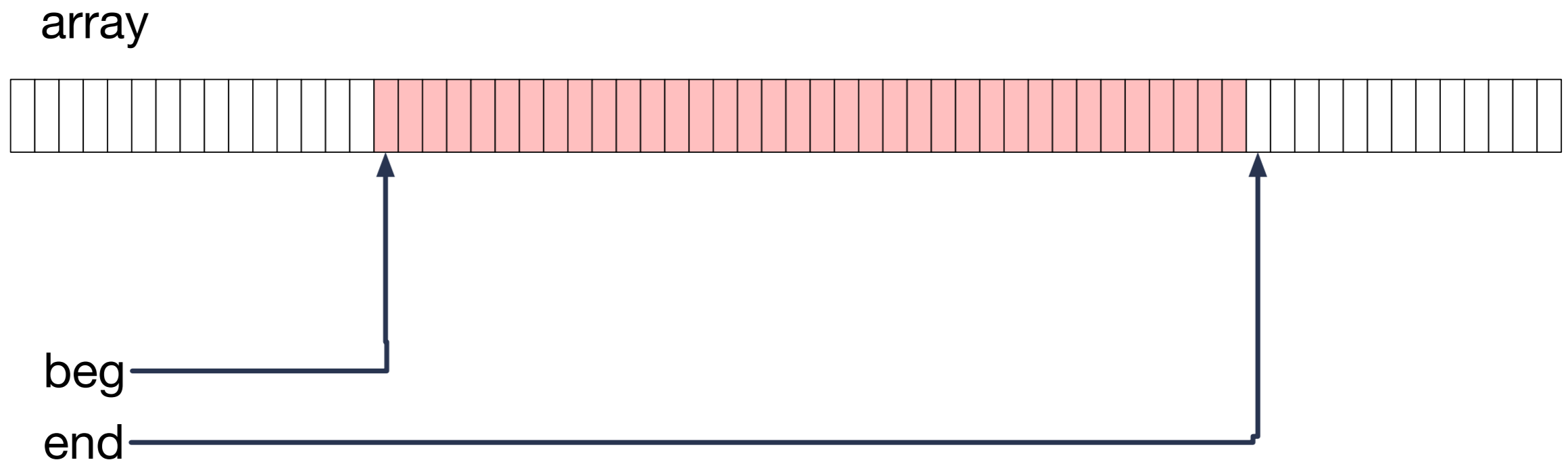
- This way, multiplication of m -bit numbers takes $m^{1.58496}$ bit multiplications

Integer Multiplication

- Can be used for arbitrary length integer multiplication
- Base case is 32 or 64 bits
- But can still do better using Fast Fourier Transformation

Binary Search

- Given an array of ordered integers, a pointer to the beginning and to the end of a portion of the array, decide whether an element is in the slice
- `Search(array, beg, end, element)`



Binary Search

- Divide: Determine the middle element. This divides the array into two subsets
- Conquer: Compare the element with the middle element. If it is smaller, find out whether the element is in the left half, otherwise, whether the element is in the right half
- Combine: Just return the answer to the one question

Binary Search

```
def binary_search(array, beg, end, key):  
    if beg >= end:  
        return False  
    mid = (beg+end)//2  
    if array[mid]==key:  
        return True  
    elif array[mid] > key:  
        return binary_search(array, beg, mid, key)  
    else:  
        return binary_search(array, mid+1, end, key)
```

```
test = [2, 3, 5, 6, 12, 15, 17, 19, 21, 23, 27, 29,  
        31, 33, 35, 39, 41]  
print(binary_search(test, 0, len(test), 21))  
print(binary_search(test, 0, len(test), 22))
```

Binary Search

- Let $T(n)$ be the runtime of `binary_search` on a subarray with n elements
- Recurrence: There is a constant c such that

$$T(1) \leq c$$
$$T(n) \leq T(n//2) + c$$

Binary Search

- Solving the recursion

$$\begin{aligned}T(n) &\leq T(n//2) + c \\ &\leq T(n//4) + 2c \\ &\dots \\ &\leq T(n//2^m) + mc\end{aligned}$$

- If $m \geq \log_2 n$ then $T(n) \leq T(1) + mc = (m + 1)c$

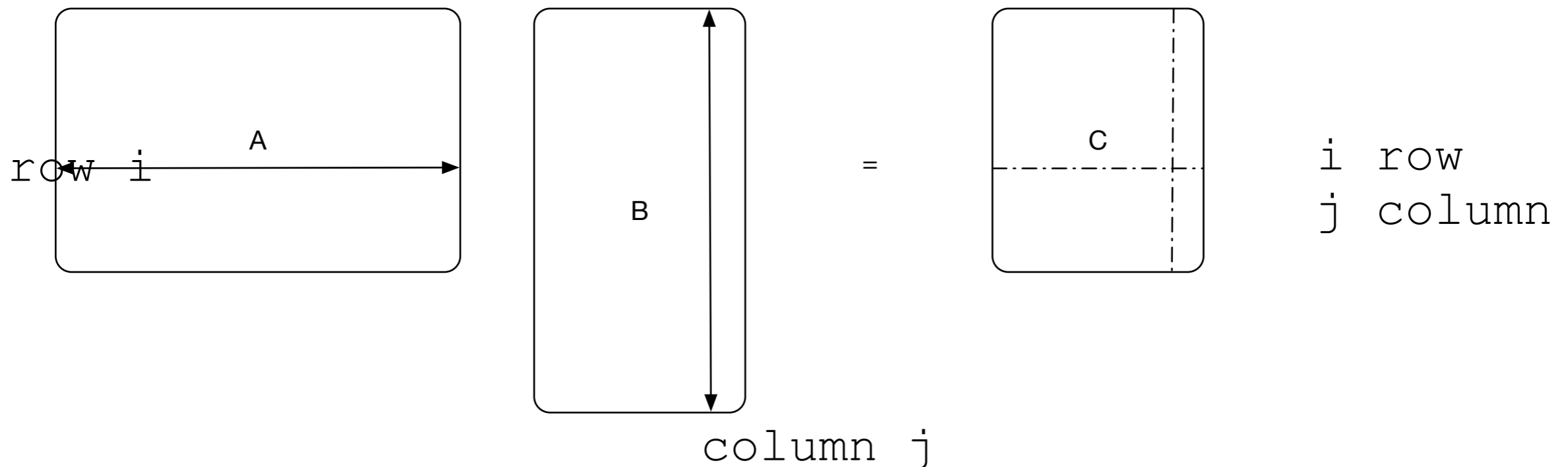
Binary Search

- With other words, binary search on n elements takes time
 $\propto \log_2(n)$

Strassen Multiplication

- Definition of Matrix Multiplication

$$\bullet \left(a_{i,j} \right)_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}} \cdot \left(b_{j,k} \right)_{\substack{1 \leq j \leq n \\ 1 \leq k \leq p}} = \left(\sum_{j=1}^n a_{i,j} b_{j,k} \right)_{\substack{1 \leq i \leq m \\ 1 \leq k \leq p}}$$



Strassen Multiplication

- Cost of definition:
- n multiplications for all n^2 elements in the product
 - Square $n \times n$ matrices: n^3 multiplications

Strassen Multiplication

- Divide and conquer: Assume $n = 2^r$ is a power of two.
- We can use the following theorem:

- Break each matrix into four submatrices of size $2^{r-1} \times 2^{r-1}$ and calculate

- $$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{21} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{21} + A_{22}B_{22} \end{pmatrix}$$

Strassen Multiplication

- As is, a divide and conquer algorithm gives us 8 multiplication of matrices half the size.
- Let $m(n)$ be the number of multiplications needed to multiply two $2^n \times 2^n$ matrices using divide and conquer
- Obviously: $m(0) = 1$
- Recursion: $m(n + 1) = 8m(n)$

Strassen Multiplication

- Claim: $m(n) = 2^{3n}$
- Proof: Induction base: $m(0) = 1 = 2^{3 \cdot 0}$
- Induction step:
 - Hypothesis: $m(n) = 2^{3n}$
 - To show: $m(n + 1) = 2^{3(n+1)}$
 - Proof:

$$m(n + 1) = 8m(n) = 8 \cdot 2^{3n} = 2^3 \cdot 2^{3n} = 2^{3n+3} = 2^{3(n+1)}$$

Strassen Multiplication

- That is the same as the normal algorithm!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Strassen Multiplication

- Strassen: Can use 7 matrix multiplications to calculate all eight products
 - $\mathbf{M}_1 := (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2})$
 - $\mathbf{M}_2 := (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1}$
 - $\mathbf{M}_3 := \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2})$
 - $\mathbf{M}_4 := \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1})$
 - $\mathbf{M}_5 := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2}$
 - $\mathbf{M}_6 := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})$
 - $\mathbf{M}_7 := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})$

Strassen Multiplication

- Then can get all the submatrices on the right:
- $\mathbf{C}_{1,1} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$
- $\mathbf{C}_{1,2} = \mathbf{M}_3 + \mathbf{M}_5$
- $\mathbf{C}_{2,1} = \mathbf{M}_2 + \mathbf{M}_4$
- $\mathbf{C}_{2,2} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$

Strassen Multiplication

- Now the recurrence becomes
 - $m(n + 1) = 7m(n), \quad m(0) = 1$
- which is obviously solved by
 - $m(n) = 7^n.$

Strassen Multiplication

- Remember that the size of the matrix was $2^n \times 2^n$.
- Thus, if $M(n)$ is the number of multiplications for an $n \times n$ matrix with power of 2 rows, then

- $M(n) = m(\log_2(n)) = 7^{\log_2(n)}$

- Since

$$\log_2(7^{\log_2(n)}) = \log_2(n)\log_2(7) = \log_2(7)\log_2(n) = \log_2(n^{\log_2(7)})$$

- $M(n) = n^{\log_2(7)} \approx n^{2.80735}$

Strassen Multiplication

- The algorithm can be extended for matrices that
 - have number of rows = number of columns not a power of 2
 - are not square

Merge-Sort

- Idea:
 - It is easy to create a single sorted array out of two sorted arrays
 - Look at the first elements in each array
 - Move the smaller one into the target array

Merge-Sort

```
def merge(arr1, arr2):
    target = [ ]
    ione, itwo = 0,0
    while ione<len(arr1) and itwo<len(arr2):
        if arr1[ione]<arr2[itwo]:
            target.append(arr1[ione])
            ione += 1
        else:
            target.append(arr2[itwo])
            itwo += 1
    if ione == len(arr1):
        target += arr2[itwo:]
    else:
        target += arr1[ione:]
```

Merge-Sort

- Example
 - Merge

0	1	5	8	11	12	13
---	---	---	---	----	----	----

- | | | | | | | |
|---|---|---|---|---|---|----|
| 2 | 3 | 4 | 6 | 7 | 9 | 10 |
|---|---|---|---|---|---|----|

- Initialize target list, set two indices equal to 0

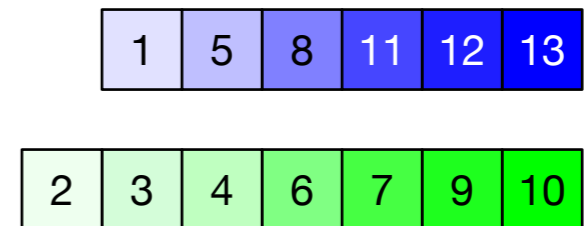
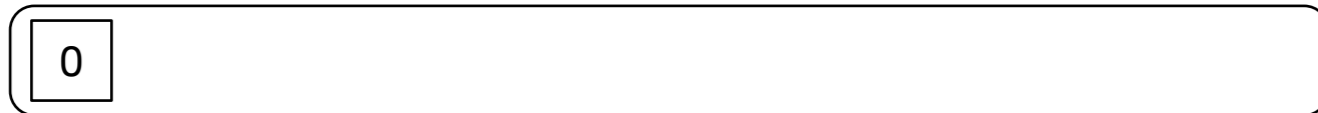
Merge-Sort

- Compare elements at indices



-

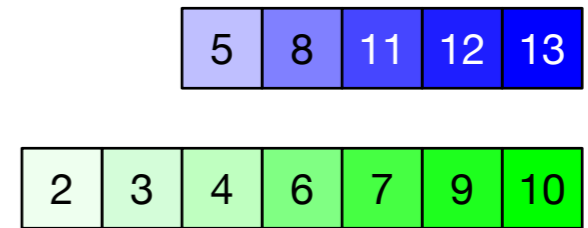
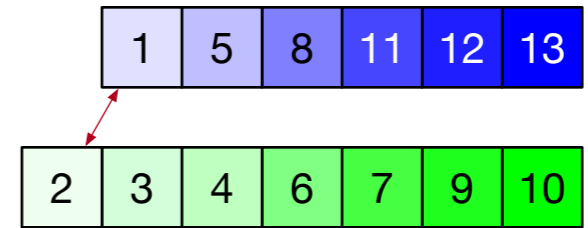
- $0 < 2$: Select 0 and move first index to right



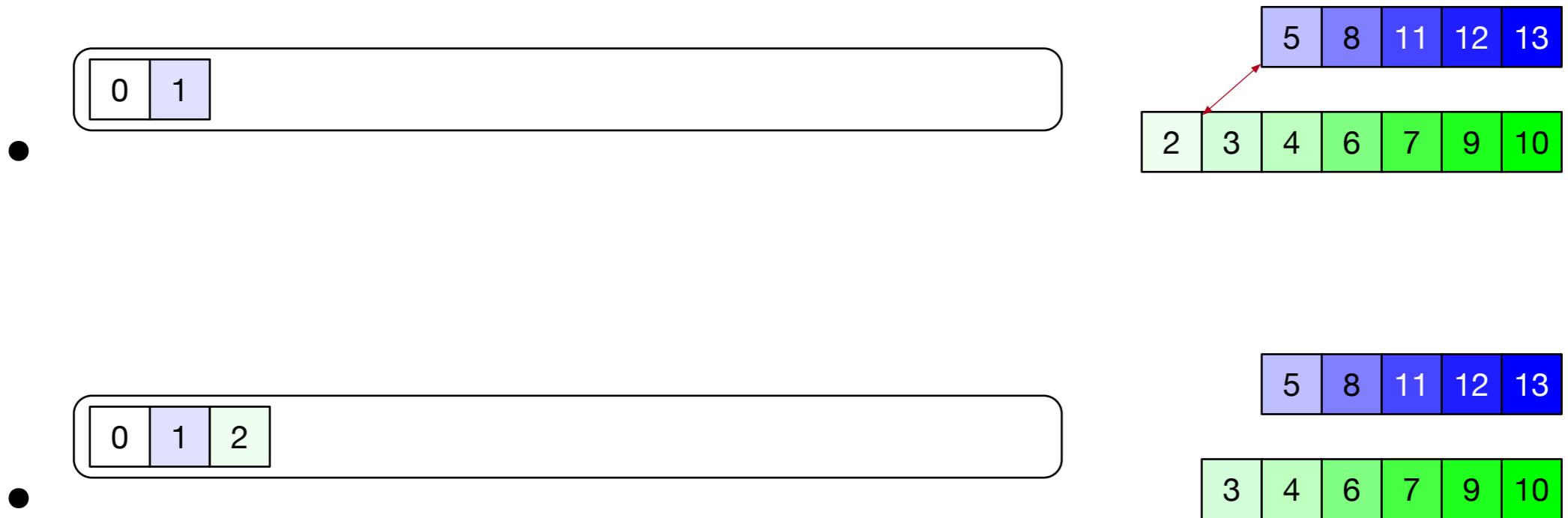
-

Merge-Sort

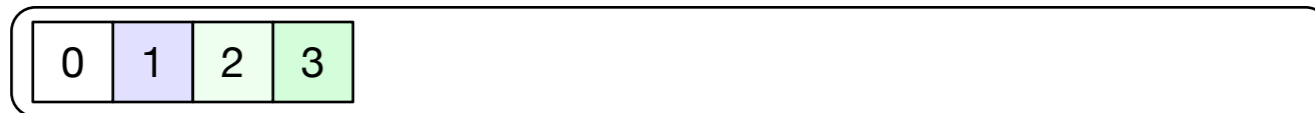
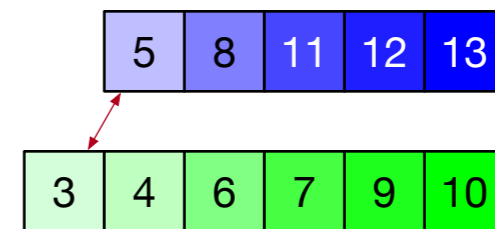
- Repeat



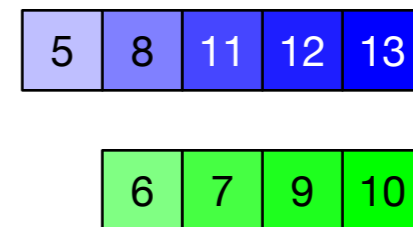
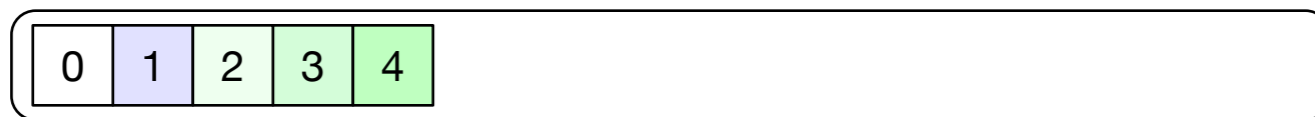
Merge-Sort



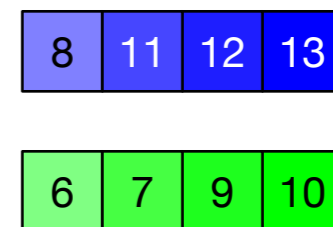
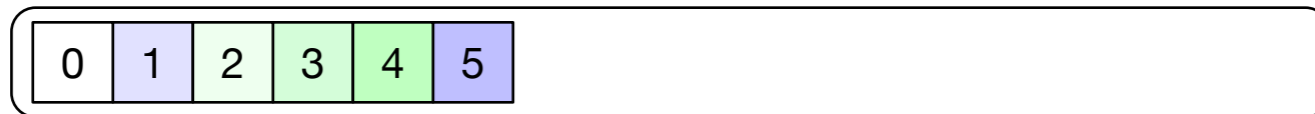
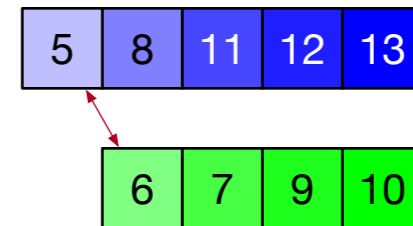
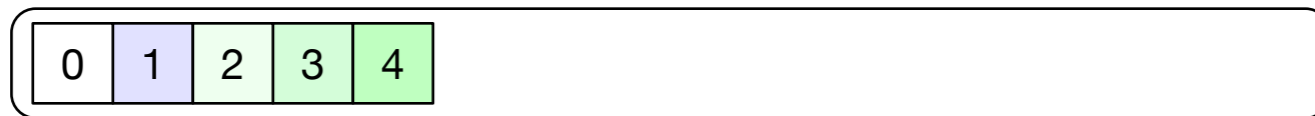
Merge-Sort



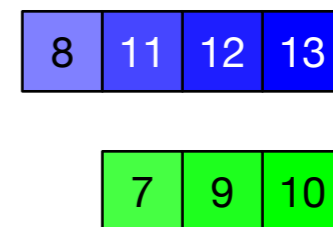
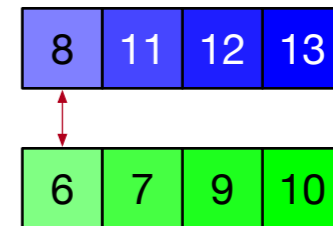
Merge-Sort



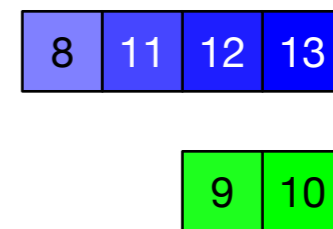
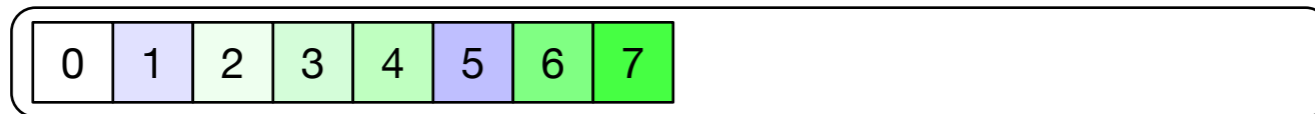
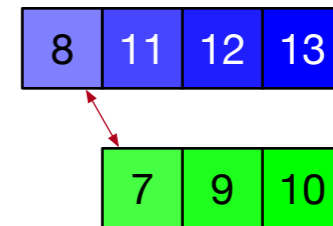
Merge-Sort



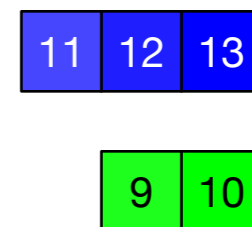
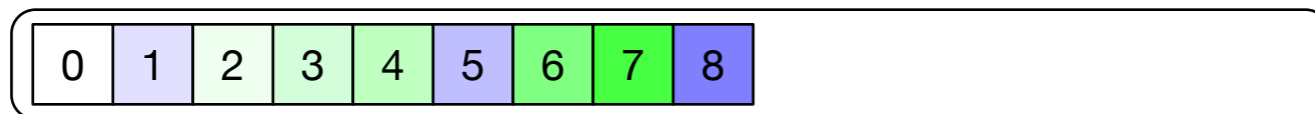
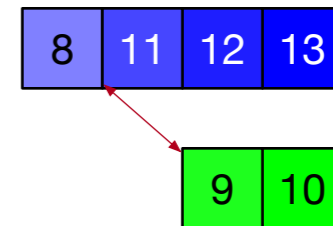
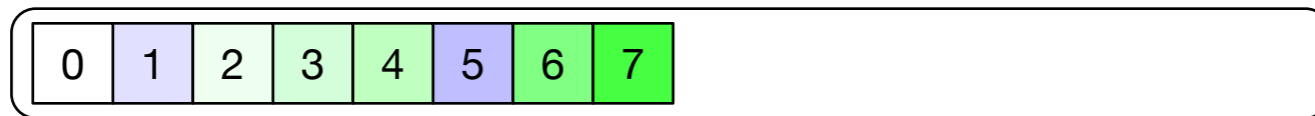
Merge-Sort



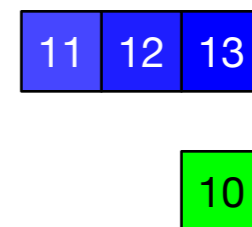
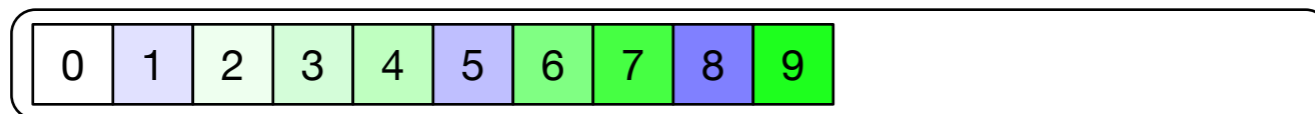
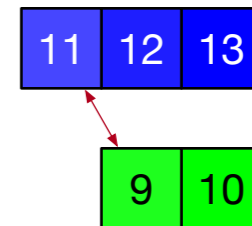
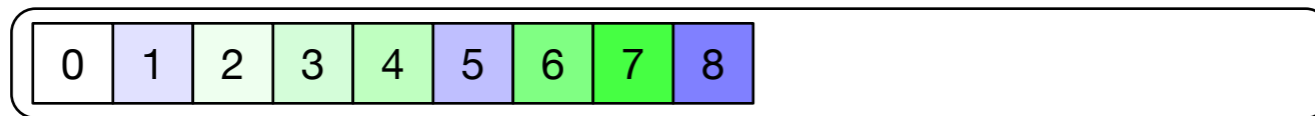
Merge-Sort



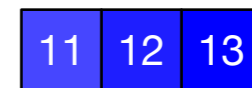
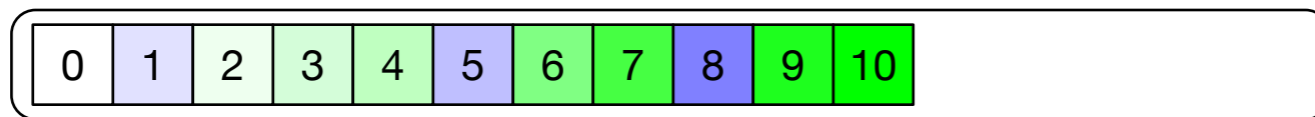
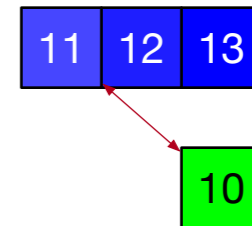
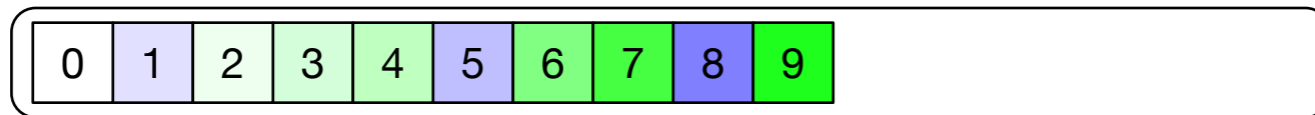
Merge-Sort



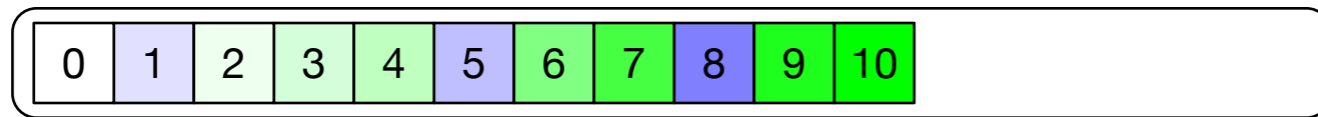
Merge-Sort



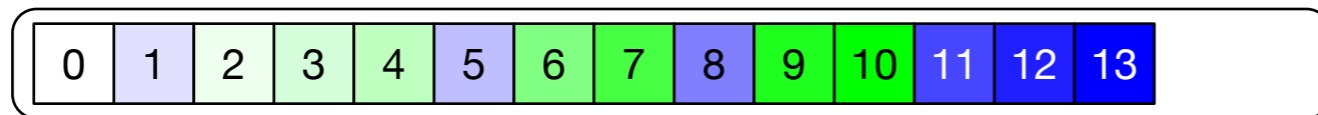
Merge-Sort



Merge-Sort



Second Index has reached the end of array: Expand with first



Merge-Sort

- Divide and conquer:
 - Divide array in two halves
 - ```
mid = len(arr) // 2
arr1, arr2 = arr[:mid], arr[mid:]
```
  - Apply recursively merge-sort
    - ```
arr1 = merge_sort(arr1)
arr2 = merge_sort(arr2)
```
- Merge both arrays

Merge-Sort

```
def merge_sort(arr):  
    if len(arr) < 2:  
        return arr  
    mid = len(arr)//2  
    arr1, arr2 = arr[:mid], arr[mid:]  
    arr1 = merge_sort(arr1)  
    arr2 = merge_sort(arr2)  
    return merge(arr1, arr2)
```

Merge-Sort

- In practice:
 - Merge-sort is not so good on very small arrays
 - Use something as bad as bubble-sort for arrays of small size

Merge-Sort

- Performance:
 - Merge of two arrays with $n_1 + n_2 = n$ elements total?
 - Up to $n - 1$ comparisons
 - Recurrence formula for the number of comparisons is approximately
 - $C(n) = 2 \cdot C(n/2) + n$

Merge-Sort

- Ad hoc solution of the recurrence relation
 - $C(n) = 2C(n/2) + n$
 - $= 2 \cdot (2C(n/4) + \frac{n}{2}) + n = 4C(n/4) + n + n$
 - $= 8C(n/8) + n + n + n$
 - $= 16C(n/16) + n + n + n + n$
 - ...
 - $= n + n + \dots n = \log(n)(n + 1)$

Quick-Sort

- Merge Sort:
 - Divide is simple
 - Work is done in the merge step
- Quick Sort
 - Work is done in the divide step
 - Conquer part is simple
 - Key Idea:
 - Pick a pivot, form two arrays: those smaller than the pivot and those larger than the pivot