**Order of Big-O Functions**

|  |  |  |
| --- | --- | --- |
| *Type* | *Function* | *Useful Facts* |
| Logarithmic | log n | log2(n) > log3(n) > log4(n) |
| Poly Logarithmic | (log n)a | (log n)2 < (log n)3 < (log n)4 |
| Fractional Power | na *where a (0..1)* | n0.2, |
| Linear | n | 2n, |
| Linear Logarithmic (*n log n*) | n log(n) | 5n log n |
| Polynomial | nb *where b > 1* | n2, n3 |
| Exponential | an | 2n, 3n+1 |
| Linear Exponential | nn | n2n |
|  |  |  |

**Math Facts**

*Exponents*

|  |  |
| --- | --- |
| ***Rule*** | ***Explanation*** |
|  | Same base? Can add exponents. |
|  | Same exponent? Can multiply base. |

*Logs*

|  |  |
| --- | --- |
| ***Rule*** | ***Explanation*** |
|  | For a base b > 1 and *any* exponent a > 0, this inequality holds for large enough *n*  *E.g.* or |
|  |  |
|  | The smaller the base, the larger the log. The difference between log bases is just a *constant* |
|  | Quotient rule. |

Cheat Sheet

*Logs*

for a base b >  1 and any exponent a > 0, we have that logbn na for large enough n (n k for some constant k)

|  |  |  |
| --- | --- | --- |
| **function** | **name** | **Asymptotic order** |
| (1) | Constant |  |
| (log log n) | Log log |  |
| (log n) | Logarithmic | log n, logcn where c is some constantlog n  n, n2, n3, etc for all n 1  inverse of 2n is log n  log n can be replaced with n for some n k  logn na for large enough n  log (nx) = x logn = lognx --> logn2=2logn  O(logn)=O(logn) |
| (log n)x | Poly Logarithmic | log n2log n3log n4.... |
| (nx for 0 < x < 1)  n | Fractional | n0.1 ≤ n0.2  ≤ n0.3...  n=n1/2 |
| (log2 n) | Log squared | log2n = logn \* logn = (logn)2 |
| (n) | Linear | n |
| (n log n) | n log n | nlogn ≤ n(logn)2 ≤ n(logn)3  log(n!) is (n log n) |
| (n2) | Quadratic |  |
| (n3) | Cubic |  |
| (nk) for k > 1 | Polynomial | n2n3n4... |
| (2n) | Exponential | 1.5n2n3n... |
| (n!) | Factorial |  |
| (nn) | Super exponential |  |

(log n)2 << n.1 << n.4 << n,5 << n << n2

Text

Description automatically generated

logab = logcb / logca

Example: to get from log2n to log3n, you need to multiply it by 1 / log32 → log2n = log3n / log32

Table

Description automatically generated

Table

Description automatically generated

Chart

Description automatically generated

Shape, rectangle

Description automatically generated

A picture containing diagram

Description automatically generated    Letter

Description automatically generated

*Runtimes*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Merge (r)** | **Quick (r)**  **rand piv** | **Quick (r) last piv** | **Insertion** | **Bubble** | **Selection** | **Heap** | **Bucket** | **Radix** | **Counting** |
| sorted | nlogn | nlogn | n2 | n | n | n2 | nlogn | n + k | d(n+r) | n + m |
| reverse sorted | nlogn | nlogn | n2 | n2 | n2 | n2 | nlogn | n2 (clustered) | d(n+r) | n + m |
| random | nlogn | nlogn | nlogn | n2 | n2 | n2 | nlogn | n + k | d(n+r) | n + m |
| stable | no | no | no | no | yes | no | no | y if underlying sort is stable | yes | yes |
| made stable | yes | yes | yes | yes |  | yes | yes |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Binary search (r)** | **Select** | **Random Select** | **Heap I** | **Heap BU** | **Heap Insert** | **Heap Delete Max** |
| best | 1 | n | n | n | n | logn | logn |
| worst | logn | n | n2 | nlogn | n |  |  |
| random | logn | n | n |  |  |  |  |

*Recur/Run we know from class*

|  |  |
| --- | --- |
| **O (log n):** |  |
| T(n) = T(n/2) + c [week2, p2] | Binary Search |
| T(n) = T(n/3) + c [a1, p2d] | Tertiary Search |
| T(n) = T(√n) + logn [week2, p8] |  |
| **O (n):** |  |
| T(n) = T(n/2) + cn [week3, p6] | Finding the median in constant time still runs Select in linear time. |
| T(n) = 2T(n/2) + c [week2, p3] | recursive calls on ½ array & constant ops: Findmax w/1 midpoint |
| T(n) = 3T(n/3) + c [week2, p4] | recursive calls on ⅓ array & constant operations: Findmax w/2 midpoints |
| T(n) = c(n-1) + d [week1, p1] | Best case runtime of Insertion Sort |
| T(n) = T(n/4) + T(n/2) + cn [a1, p2a] | Uneven recursion split of array. (1) ¼ recursion, (1) ½ recursion |
| T(n) = 2T(n/4) + n [week2, p8] |  |
| T(n) = T(n/5) + T(3/4n) + cn [week3, order stats note] | Select algorithm |
| **O(n log n):** |  |
| T(n) = T(n-1) + clogn [week2, p6] | recursive call on array size of 1 less & binary search operation (log n) |
| T(n) = T(n/3) + T(2n/3) + O(n) [week3, p9] | Select w/groups of 3 instead of 5 |
| T(n) = 3T(n/3) + O(n) [week5, p2] | Quicksort w/2 partitions |
| T(n) = 2T(n/2) + O(n) [week5, p3] | Mergesort |
| **O(n2)**: |  |
| T(n) = T(n-1) + cn [week2, p5] | Insertion Sort (recursive) |
| T(n) = T(n-2) + cn [week3, p10] | Rand-select: picks 3 random numbers and uses median of 3 as pivot |
| T(n) = T(n-3) + cn [a1, 4c] | Rand-Select when pivot is always 3rd smallest |
| T(n) = an2 + bn + c [week1, p3] | Selection Sort |

*Pseudocode*

Insertion Sort (Iterative): Insertion Sort (Recursive):

for i=2 to n, do: if s < f //array is not just size 1

j=1       InsertionSort(A,s,f-1)

while j > 1 j=f //insert last element

if A[j-1] > A[j] while j > 1

swap A[j-1] and A[j]               if A[j-1] > A[j]

j = j-1   swap A[j-1] and A[j]

else break   j = j-1

  else break

Selection  Sort:

for i=1 to n-1:

min = find\_min(A,i+1,n)

swap A[i] and A[min]

Mergesort:

Mergesort (A,s,f) //Array, s = index start, f = index finish

 if s < f //Make sure the array has more than one element in it

    q = floor[(f+s)/2] //Split position

    Mergesort (A,s,q) //Call Mergesort from start to split position

    Mergesort (A,q+1,f) //Call Mergesort from split position + 1 to finish

    Merge(A,s,q,f) //Merge lists together

Quicksort:

Quicksort (A,s,f)

 if s = f return //Array has size one

 r = Partition(A,s,f)

 Quicksort (A,s,r-1,f)

 Quicksort (A,r+1,f)

Partition(A,s,f)

slide = s-1

for j=s to (f-1)

 if A[j] < A[f]

 slide = slide+1

 swap A[j] with A[slide]

swap A[f] with A[slide+1]

return slide+1 //returns rank of pivot

Bubble-up & Bubble-down:

Graphical user interface, text, application

Description automatically generated

Binary  Search:

BSearch (A,s,f,k)

 if (s < f)

q = round-down((f+s)/2)

if A[q] = k

 return true

else if A[q] < k

return BSearch(A,q+1,f)

else

Return BSearch(A,s,q-1)

 else if A[s] = k, return true

 else return false

Find  Max:

Findmax (A,s,f) Findmax (A,s,f)

 if (s < f)       if (s < f)

q = round-down((f+s)/2) q1 = first third

m1 = Findmax (A,s,q) q2 = second third

m2 = Findmax (A,q+1,f) m1 = Findmax (A,s,q1)

if (m1 > m2) return m1 m2 = Findmax (A,q+1,q2)

else return m2 m3 = Findmax (A,q2,qf)

 else return A[s] return max(m1,m2,m3)

else return A[s]

Partition for Rand-Select

Partition (A,s,f)

 p = random number between s and f inclusive

 n = f-s+1

 initialize B[1...n]

 i=1

 j=n

 for count = s to f, do:

if count p

if A[count] < A[p]

  B[i] = A[count]

  i=i+1

else

  B[j]=A[count]

  j=j-1

  B[i]=A[p]