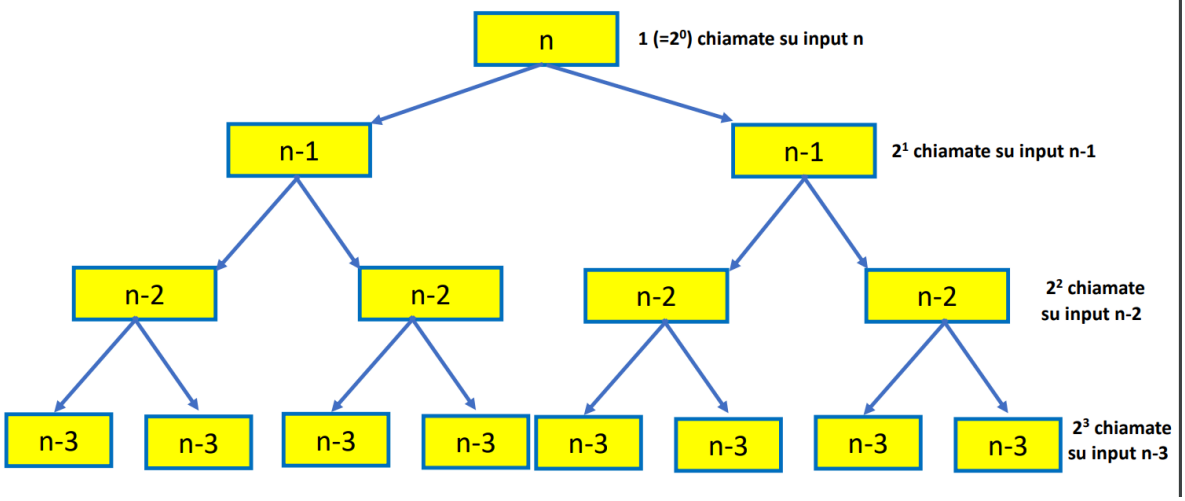
TORRE DI HANOI

```
void move(int n, char *A, char *B, char *C) {
        if (n == 1) printf("moving disc %d from tower %s to tower %s\n",n,A,C);
        else {
                move(n-1,A,C,B); /* da A a B usando C come supporto */
                printf("moving disc %d from tower %s to tower %s\n",n,A,C);
                move(n-1,B,A,C); /* da B a C usando A come supporto */
        }
}
```

Complessità in tempo $O(2^n)$

Complessità in spazio O(n)



MERGESORT

```
void MergeSort(int A[ ], int p, int r) {
        int q;
        if (p < r) {
                q = (p+r)/2;
                MergeSort(A, p, q);
                MergeSort(A, q+1, r);
                Merge(A, p, q, r);
        }
}

void Merge(int A[ ], int p, int q, int r) {
        int B[ ], i, j, k;
        i=p; j=q+1; k=p;
        B = (int *)malloc(…);
        while(i<=q && j<=r) {
                if (A[i] < A[j]) B[k++] = A[i++];
                else B[k++] =A[j++];
        }
        while(i<=q) B[k++] = A[i++];
```

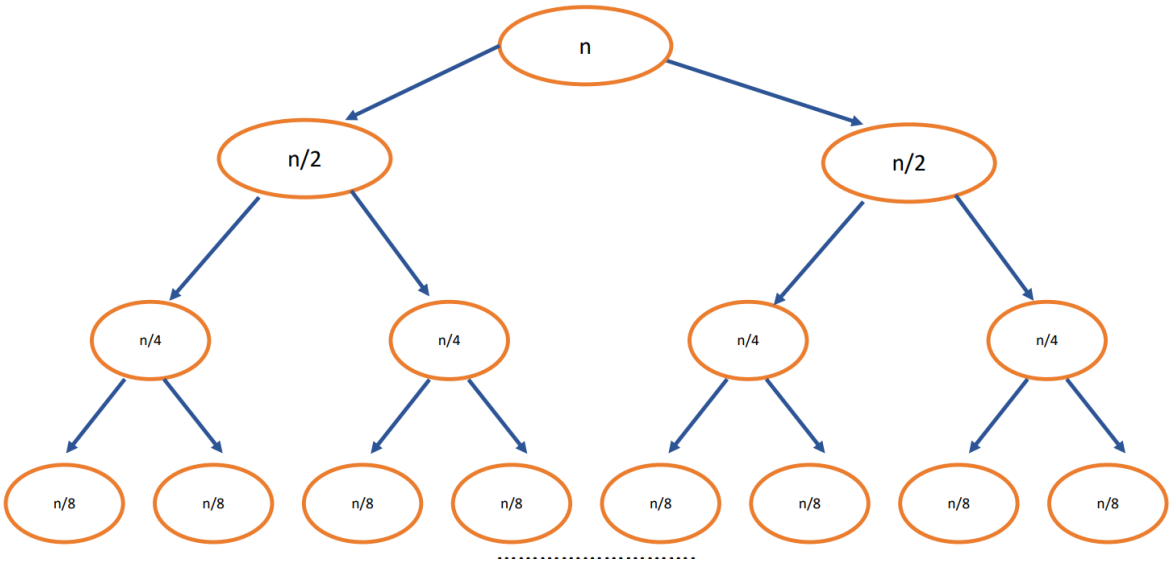
```
        while(j<=r) B[k++] = A[j++];
        for (k=p; k<=r; k++) A[k] = B[k];
        free(B);
}
```

Complessità in tempo O(n*$log_2 n$)

-> Complessità della Merge O(k) k=r-p+1;

Ad ogni livello facciamo complessivamente n confronti!
Essendo l'altezza dell'albero **log_2(n)**...

Complessità in spazio O($log_2 n$)



QUICKSORT

```
void Quicksort(int A[ ], int p, int r) {
        int q;
        if (p < r) {
                q = Partition(A[],p,r);
                QuickSort(A, p, q-1);
                QuickSort(A, q+1, r);
        }
}

int Partition(int A[ ], int p, int r) {
        int i, pivot, pivotpos;
        Swap(p,(p+r)/2,A);
        pivot= A[p];
        pivotpos= p;
        for(i=p+1; i<= r; i++)
                if(A[i]<pivot) Swap(++pivotpos, i, A);
                Swap(p, pivotpos, A);
                return(pivotpos);
        }
```

```
void Swap(int i, int j, int A[]) {
        int tmp;
        tmp = A[i];
        A[i] = A[j];
        A[j] = tmp;
}
```

Complessità in tempo

Caso migliore O(n*$log_2 n$)

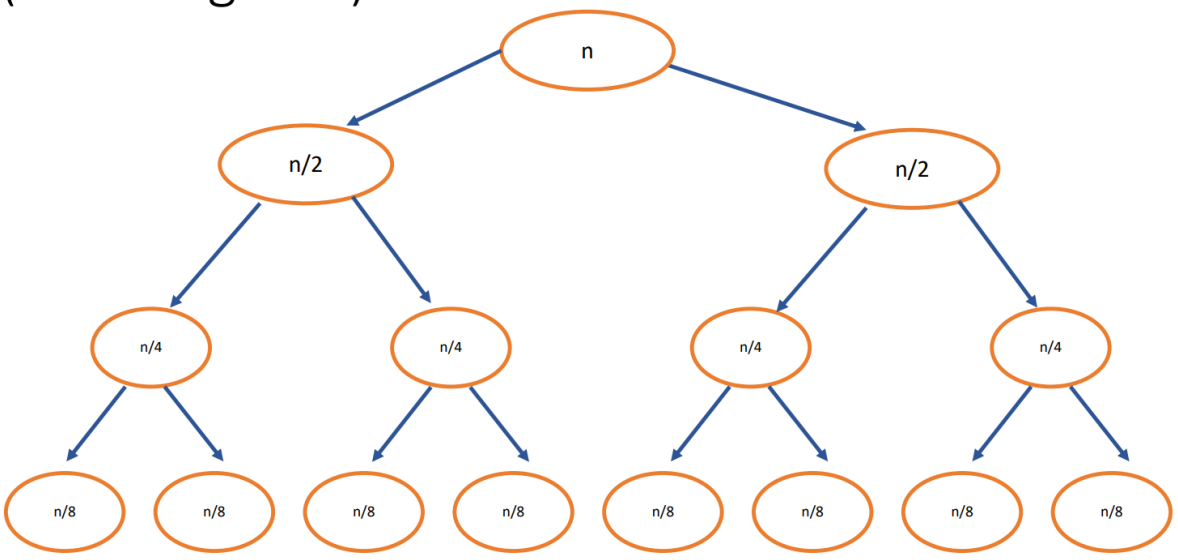
Caso peggiore O(n)

Complessità in spazio

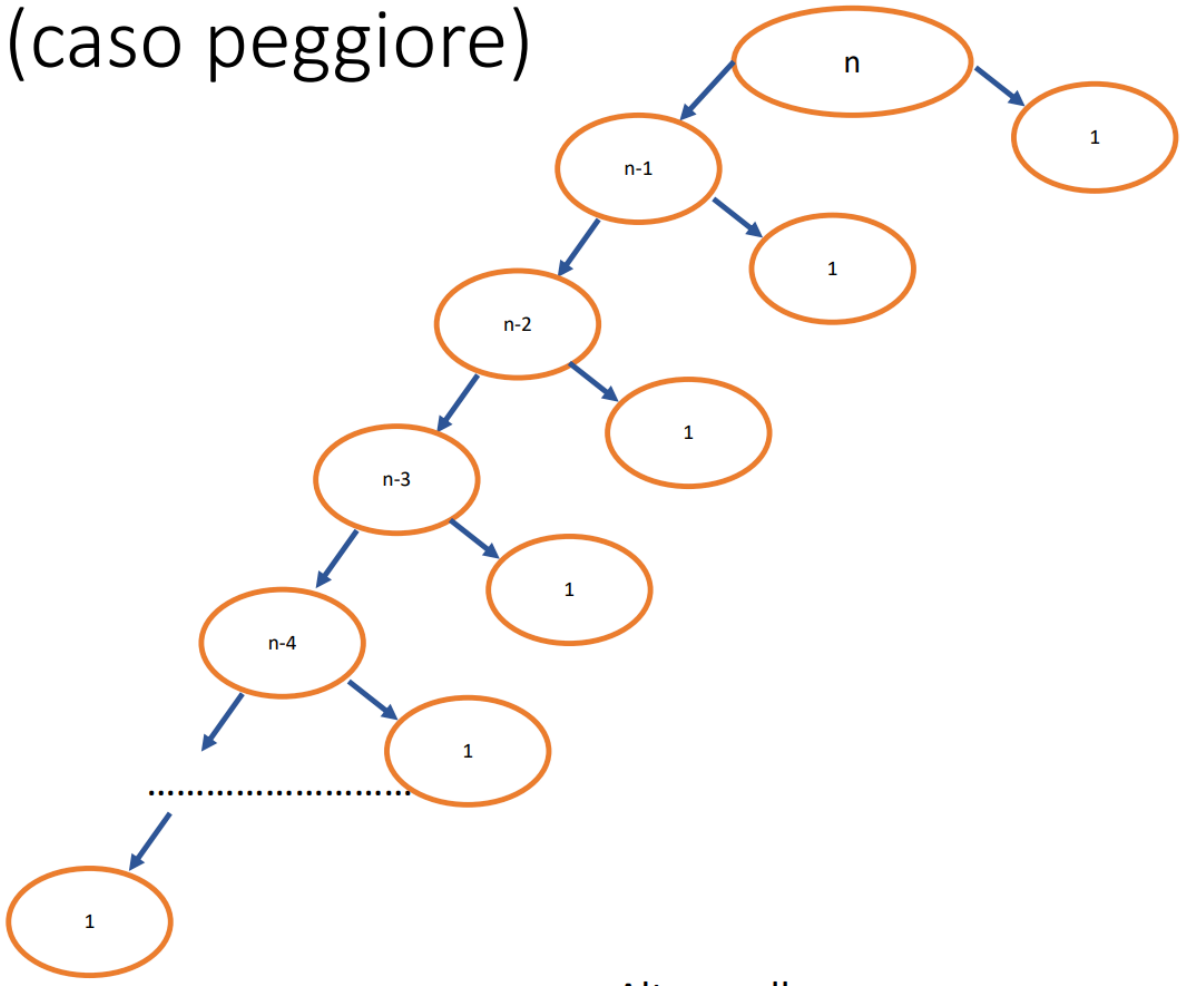
- CASO MIGLIORE
  - Altezza albero, $\log_2(n)$
  - al massimo avremo $\log_2(n)$ record di attivazione sullo stack
  - Complessità $O(\log_2(n))$

- CASO PEGGIORE
  - Altezza albero, n
  - al massimo avremo n record di attivazione sullo stack
  - Complessità O(n)

(caso migliore)



(caso peggiore)



Altezza albero **n**

```
struct nodo* build_list2(int v[], int i){                    //RICORSIVA
    struct nodo* head;
    if(v[i]==-1)
        return NULL;

    head=(struct nodo*)malloc(sizeof(struct nodo));
    head->data=v[i];
    head->next=build_list2(v, i+1);
    return head;
}
```

```
struct nodo* build_list1(int v[]){
    struct nodo* head=NULL;
    struct nodo* tail=NULL;
    struct nodo* tmp=NULL;
```

```
    head=(struct nodo*)malloc(sizeof(struct nodo));
    head->data=v[0];
    head->next=NULL;
    tail=head;

    for(int i=1; v[i]!=-1; i++){
        tmp=(struct nodo*)malloc(sizeof(struct nodo));
        tmp->data=v[i];
        tmp->next=NULL;
        tail->next=tmp;
        tail=tmp;
    }
    return head;
}
```

```
void print_list(struct nodo* head){
    if(head==NULL){
        printf("NULL\n");
        return;
    }

    printf("%d -> ", head->data);
    print_list(head->next);
}
```

```
void invert_list(struct nodo** head){              //iterativa
    struct nodo* corrente=*head;
    struct nodo* successivo=NULL;
    struct nodo* precedente=NULL;

    while(corrente!=NULL){
        successivo=corrente->next;
        corrente->next=precedente;
        precedente=corrente;
        corrente=successivo;
    }
    (*head)=precedente;
}
```

```
struct nodo* invert_list(struct nodo* head, struct nodo* pred){                  //ricorsiva
    if(head==NULL)
```

```
        return pred;

    struct nodo* tmp=head->next;
    head->next=pred;

    return invert_list(tmp, head);
}
```

```
void delete_x(struct nodo** head, int x){
    if(*head==NULL)
        return;

    if((*head)->data==x){
        struct nodo* tmp=NULL;
        tmp=*head;
        *head=(*head)->next;
        free(tmp);
    }
    delete_x(&(*head)->next, x);
}
```

```
struct nodo* new_list(struct nodo* head, int x){
    if(head==NULL){
        return NULL;
    }

    if(head->data<x){
        struct nodo* tmp=(struct nodo*)malloc(sizeof(struct nodo));
        tmp->data=head->data;
        tmp->next=new_list(head->next, x);
        return tmp;
    }
    else
        return new_list(head->next, x);
}
```

```
int crescente(struct nodo* head){
    while(head->next!=NULL){
        if(head->data>=head->next->data){
            return 0;
        }
        head=head->next;
```

```
    }
    return 1;
}
```

```
int ordine(struct nodo* head1, struct nodo* head2){
    if(head1==NULL)
        return 1;
    if(head1!=NULL&&head2==NULL)
        return 0;

    if(head1->data==head2->data)
        return ordine(head1->next, head2->next);
    else
        return ordine(head1, head2->next);
}
```

```
struct brano* build_playlist(char **titolo, char **autore, int *durata, int n, int i){
    struct brano* head=(struct brano*)malloc(sizeof(struct brano));
    if(i==n)
        return NULL;

    if(head==NULL)
        exit(1);

    head->titolo=titolo[i];
    head->autore=autore[i];
    head->durata=durata[i];
    head->next=build_playlist(titolo, autore, durata, n, i+1);
    return head;
}
```

```
void headisert (struct nodo** head, int x){
    struct nodo* new_node = (struct nodo*) malloc(sizeof(struct nodo));
    if(new_node==NULL){
        fpritnf(stderr, "Could not allocate memory!");
        return;
    }

    new_node->data=x;
    new_node->next=*head;
    *head=new_node;
```

```
}
```

```
void tailinsert(struct nodo* head, int x){
   struct nodo* new_node = (struct nodo*) malloc(sizeof(struct nodo));
   if(new_node==NULL){
      fpritnf(stderr, "Could not allocate memory!");
      return;
   }
   new_node->data=x;
   new_node->next=NULL;

   while(head->next!=NULL)
      head=head->next;

   head->next=new_node;
}
```

MAKEFILE

```
CC=gcc
CFLAGS=-Werror -Wpedantic -Wextra
DEPSS= Liste.h

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

build: main.o Liste.o
	$(CC) -o main.exe main.o Liste.o $(CFLAGS)

clean:
	rm *.o
```

FILE .h

```
#ifndef _LISTS_H
#define _LISTS_H

Prototipi funzioni

#endif
```